Analyzing Data from Facebook, Twitter, LinkedIn, and Other Social Media Sites

2nd Edition

Early Release

RAW & UNEDITED

Mining the Social Web

O'REILLY®

Matthew A. Russell

SECOND EDITION

Mining the Social Web

Matthew A. Russell

O'REILLY®
Beijing • Cambridge • Farnham • Köln • Sebastopol • Tokyo

Mining the Social Web, Second Edition

by Matthew A. Russell

Printed in the United States of America.

Published by O'Reilly Media, Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472.

Proofreader: Marlowe Shaeffer **Illustrator:** Robert Romano
Interior Designer: David Futato

Revision History for the :

2013-05-03: Early release revision 1

2013-05-09: Early release revision 2

See *http://oreilly.com/catalog/errata.csp?isbn=9781449367619* for release details.

ISBN: 978-1-449-36761-9

[LSI]

Use wisdom and understanding to establish your home;
Let good sense fill the rooms with priceless treasures.
Wisdom brings strength, and knowledge gives power.
Battles are won by listening to advice and making a lot of plans.

Table of Contents

Preface ..... **ix**

Part I. A Guided Tour of the Social Web

1. Mining the Twitterverse: Exploring Trending Topics, Discovering What People Are Talking About, and More **3**
The Twitter Developer Platform 4
Making Twitter API Requests 6
Exploring Trending Topics 9
Searching for Tweets 12
Basic Anatomy of a Tweet 18
Discovering What People Are Talking About 20
Visualizing Frequency Data with Histograms 27
Closing Remarks 31
Recommended Exercises 32

2. Mining Facebook's Social Graph: Analyzing Fan Pages, Examining Friendships, and More **35**
The Facebook Platform 36
Exploring the Graph API 37
Open Graph Protocol 43
Accessing the Graph API 47
Exploring the Graph API one connection at a time 53
Analyzing Facebook Pages 57
Examining Friendships 62
Constructing and Analyzing Mutual Friendship Graphs 69
Visualizing Mutual Friendships with Force Directed Graphs 73
Closing Remarks 75

Recommended Exercises 76

3. Mining Your LinkedIn Professional Network: Faceting Job Titles, Clustering Colleagues, and More. 77
The LinkedIn Developer Network 78
Making LinkedIn API Requests 79
Prerequisite Knowledge for Clustering 84
Normalizing LinkedIn Data 86
Normalizing and Counting Companies 86
Normalizing and Counting Job Titles 88
Normalizing and Counting Locations 91
Visualizing Locations with Cartograms 94
Measuring Similarity 96
Approaches to Clustering 101
Greedy Clustering 102
Hierarchical Clustering 109
K-Means Clustering 113
Clustering Your Professional Network with K-Means and Visualizing it with Google Earth 116
Closing Remarks 121
Recommended Exercises 121

4. Mining Google+ Activities: Computing Document Similarity, Extracting Collocations, and More. 123
The Google+ Platform 124
Exploring the Google+ API 126
Exploring Human Language Data with the Natural Language Toolkit 134
A Whiz-Bang Introduction to TF-IDF 139
Querying Google+ Data with TF-IDF 146
Finding Similar Documents 148
The Theory Behind Vector Space Models and Cosine Similarity 148
Clustering Posts with Cosine Similarity 150
Visualizing Document Similarity with a Matrix Diagram 153
Bigram Analysis 154
Contingency Tables and Scoring Functions 158
Closing Remarks 162
Recommended Exercises 164

5. Mining Web Pages: Using Natural Language Processing to Understand Human Language, Summarize Blog Posts, and More. 167
Scraping, Parsing, and Crawling the Web 168
Breadth-first Search 172

Decoding Syntax to Discover Semantics 174
Sentence Detection in Human Language Data 180
Document Summarization 184
Analysis of Luhn's Summarization Algorithm 191
Entity-Centric Analysis: A Deeper Understanding of the Data 193
Quality of Analytics for Processing Human Language Data 202
Closing Remarks 205
Recommended Exercises 205

Preface

The Web is more a social creation than a technical one. I designed it for a social effect—to help people work together—and not as a technical toy. The ultimate goal of the Web is to support and improve our weblike existence in the world. We clump into families, associations, and companies. We develop trust across the miles and distrust around the corner.

—Tim Berners-Lee, *Weaving the Web* (Harper)

README.1st

The basic set of assumptions this book makes about you as a reader is that you want to learn how to mine data from popular social web properties, have lots of fun along the way with no technology hassles, and ultimately, feel as though this book has been a good investment of your money and time. As an intelligent and busy person, your time is your most valuable asset, so it's important to manage your expectations about the potential use of your time, and the remainder of this preface is designed to do just that. If you spend just a minute or two to finish reading through it, you'll either decide that this book isn't for you (very unlikely), or you'll decide that this is the book of your dreams (extremely likely.)

Managing Expectations

The minimal requirement for fully enjoying this book and all that it has to offer is that you need to be interested in the vast possiblities for mining the rich data tucked away in popular social web properties and be motivated enough to pull down a virtual machine image and follow along with the Python-based example code in IPython Notebook, a fantastic web-based tool that features all of the example code for every chapter and makes following along with it as simple as a few keystrokes. This book will teach you a few things that you'll be thankful to know and add a few tools to your toolbox

that you'll find indispensible, but perhaps more importantly, it will tell you a story and entertain you along the way. Above all else, it assumes that you're a smart person and that your time is far too valuable to waste the better part of an evening on a mundane configuration problem. If you are spending more time troubleshooting technology than using it to answer your questions, then there is a serious problem.

If you have a basic programming background and are interested in insight surrounding the opportunities that arise from mining and analyzing data from the social web, you've come to the right place. We'll begin getting our hands dirty with Twitter, Facebook, LinkedIn, Google+, and other sources of social web content after just a brief pre-ramble. I'll be forthright, however, and say upfront that one of the chief complaints you're likely to have about this book is that all of the chapters are far too short. Unfortunately, that's always the case when trying to capture a space that's evolving daily and is so rich and abundant with opportunities.

The most popular social web properties have transitioned from fad to mainstream to household names over recent years, are really changing the way we live our lives on and off the Web, and are enabling technology to bring out the best (and sometimes the worst) in us. Generally speaking, each chapter of this book interlaces slivers of the social web along with data mining, analysis, and visualization techniques to answer the following kinds of questions:

- Who knows whom, and what friends do they have in common?
- How frequently are certain people communicating with one another?
- How symmetrical is the communication between people?
- Who are the quietest/chattiest people in a network?
- Who are the most influential/popular people in a network?
- What are people chatting about (and is it interesting)?

Although this book is short, it does cover a lot of ground. Generally speaking, there's a little more breadth than depth, although where the situation lends itself and the subject matter is complex enough to warrant a more detailed discussion, there are a few deep dives into interesting mining and analysis techniques. It was written so that you could have the option of either reading it from cover to cover to get a broad primer on working with social web data, or pick and choose chapters that are of particular interest to you. In other words, each chapter is designed to be bite-sized and fairly standalone, but special care was taken to introduce material in a particular order so that the book as a whole is an enjoyable read.

Activities such as building a turn-key killer app from scratch, venturing far beyond the typical usage of visualization libraries, and constructing just about anything state-of-the-art are not within the scope of this book. You'll be really disappointed if you purchase this book because you want to do one of those things. However, just because it's not

realistic or our goal to capture the holy grail of text analytics or build a killer app a mere few hundred pages doesn't mean that this book won't empower you with tools answer compelling questions about the social web as a domain, have a lot of fun in the process, and hopefully prosper abundantly from it (somehow.)

Finally, maybe it's obvious in this day and age, but another important item of note is that this book generally assumes that you're connected to the Internet. This wouldn't be a great book to take on vacation with you to a remote location, because most of its examples require access to social web APIs.

Python-Centric Technology

This book very intentionally and specifically takes advantage of the Python programming language for all of its example code. Python's intuitive syntax, amazing ecosystem of packages that trivialize API access and data manipulation, and core data structures that are practically JSON make it an excellent teaching tool that's powerful yet also very easy to get up and running. As if that weren't enough to make it both a great pedagogical choice as well just a very pragmatic choice for mining the social web, there's IPython Notebook, a powerful interactive Python interpreter that runs in your web browser and provides a notebook-like user experience that combines code execution, code output, text, mathematical typesetting, plots, and more. It's a little bit difficult to imagine a better user experience for a learning environment because it trivializes the problem of delivering sample code that you as the reader can follow along with and execute with no hassles. Every chapter in this book has a corresponding IPython Notebook with turnkey example code that makes it a pleasure to study the code, tinker around with it, and customize it for your own purposes.

Beyond the previous explanation, this book makes no attempt to justify the selection of Python or apologize for using it, because it's a very suitable tool for the job. If you've done some programming but have never seen Python syntax, skimming ahead a few pages should hopefully be all the confirmation that you need. Excellent documentation is available online, and the official Python tutorial is a good place to start if you're looking for a solid introduction.

This book's Python source code is written in Python 2.7, the latest release of the 2.x line. (Although perhaps not trivial, tt's not too difficult to imagine using some of the automated tools to up-convert it to Python 3 for anyone who is interested.)

IPython Notebook is great, but if you're new to the Python programming world, advising you to just follow the instructions online to set your development environment all configured and ready to go would be a bit counterproductive and rude. To make your

experience with this book as enjoyable as possible, a turn-key virtual machine is available that has IPython and all of the other Python dependencies pre-installed and ready to go. All that you have to do is download the virtual machine, start the IPython Notebook server, and open your web browser. That's literally it. (Are we having fun yet?)

Whether you're a Python novice or a guru, the book's latest bug-fixed source code and accompanying scripts for building the virtual machine is available onGitHub, a very social Git repository that will always reflect the most up-to-date example code available. The hope is that social coding will enhance collaboration between like-minded folks such as ourselves who want to work together to extend the examples and hack away at interesting problems. Hopefully, you'll fork, extend, and improve the source—and maybe even make some new friends along the way.

Be social! Bookmark the official GitHub repository that maintains getting started instructions along with the latest and greatest bug-fixed source code for this book at *https://github.com/ptwobrussell/Mining-the-Social-Web-2nd-Edition/*. Follow *@SocialWebMining* , and stay connected on Facebook by liking https://www.facebook.com/Mining-TheSocialWeb.

Notes on the Second Edition

A book is a product. First editions of any product can be vastly improved upon, aren't always what customers ideally would have wanted, and can have great potential if appropriate feedback is humbly accepted and adjustments are made. This book is no exception, and the feedback and learning experience from interacting with readers and consumers of this book's sample code over the past few years has been incredibly shaping this book to be far beyond anything I could have ever designed if left solely to my own devices. I've incorporated as much of thefeedback as possible, and it mostly boils down to the theme of simplification.

Simplification and presents itself in this second edition in a variety of ways. Perhaps most notably, one of the biggest differences in this book and the previous edition is that the technology toolchain is vastly simplified. Whereas the previous version involved a variety of databases for storage depending on the data at-hand, various visualization toolkits, and assumed that readers could just figure most of the installation and configuration out by themselves by reading the online instructions, this edition goes to great lengths to present the technology dependencies as well as the source code itself in the simplest possible manner possible.

The core toolbox is just IPython Notebook and some third-party package dependencies (all of which are versioned so that updates to open source software don't cause code breakage) that comes pre-installed on a virtual machine. Visualizations are even baked

into source code packaging, execute directly from within IPython Notebook, and are consolidated down to a single JavaScript toolkit that keeps a nice and consistent user experience across the chapters. Continuing along the theme of simplification, spending less time introducing disparate technology in the book affords the opportunity to spend more time engaging in fundamental exercises in analysis. One of the recurring themes from readers in critiques of the first edition's content is that more time should have been spent in analysis and discussing the implications of the exercises (a very fair criticism indeed), and the hope is that this second edition delivers on that wonderful suggestion by augmenting existing content with additional explanations in some of the void that was left behind. In a sense, this second edition does "more with less" and delivers significantly more value to you as the reader because of it.

In terms of structural reorganization, you may notice that an additional chapter on GitHub has been added, and the two advanced chapters on Twitter from the first edition have been refactored and expanded into a collection of more easily adaptable Twitter recipes that's organized into Part II. The chapter on microformats has been folded into the final chapter of the book that's designed to be more of a forward-looking kind of cocktail discussion than an extensive collection of programming exercises.

Conventions Used in This Book

This book is extensively hyperlinked, which makes it ideal to read in an electronic format. Purchasing it as an eBook through O'Reilly also guarantees that you will get automatic updates for the book as they become available.

The following typographical conventions are used in this book:

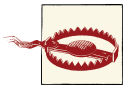
Italic
: Indicates new terms, URLs, email addresses, filenames, and file extensions.

`Constant width`
: Indicates program listings, and is used within paragraphs to refer to program elements such as variable or function names, databases, data types, environment variables, statements, and keywords.

`Constant width bold`
: Shows commands or other text that should be typed literally by the user. Also occasionally used for emphasis in code listings.

`Constant width italic`
: Shows text that should be replaced with user-supplied values or values determined by context.

This icon signifies a tip, suggestion, or general note.



This icon indicates a warning or caution.

Using Code Examples

The latest sample code for this book is maintained on GitHub at *https://github.com/ptwobrussell/Mining-the-Social-Web-2nd-Edition/*—the official code repository for this book. You are encouraged to monitor this repository for the latest bug-fixed code as well as extended examples by the author and the rest of the social coding community.

This book is here to help you get your job done. In general, you may use the code in this book in your programs and documentation. You do not need to contact us for permission unless you're reproducing a significant portion of the code. For example, writing a program that uses several chunks of code from this book does not require permission. Selling or distributing a CD-ROM of examples from O'Reilly books does require permission. Answering a question by citing this book and quoting example code does not require permission. Incorporating a significant amount of example code from this book into your product's documentation does require permission.

We appreciate, but do not require, attribution. An attribution usually includes the title, author, publisher, and ISBN. For example: "*Mining the Social Web* by Matthew A. Russell. Copyright 2013 Matthew Russell, XXX."

If you feel your use of code examples falls outside fair use or the permission given above, feel free to contact us at *permissions@oreilly.com*.

Safari® Books Online

Safari Books Online is an on-demand digital library that lets you easily search over 7,500 technology and creative reference books and videos to find the answers you need quickly.

With a subscription, you can read any page and watch any video from our library online. Read books on your cell phone and mobile devices. Access new titles before they are available for print, and get exclusive access to manuscripts in development and post feedback for the authors. Copy and paste code samples, organize your favorites, download chapters, bookmark key sections, create notes, print out pages, and benefit from tons of other time-saving features.

O'Reilly Media has uploaded this book to the Safari Books Online service. To have full digital access to this book and others on similar topics from O'Reilly and other publishers, sign up for free at *http://my.safaribooksonline.com*.

How to Contact Us

Please address comments and questions concerning this book to the publisher:

O'Reilly Media, Inc.
1005 Gravenstein Highway North
Sebastopol, CA 95472
800-998-9938 (in the United States or Canada)
707-829-0515 (international or local)
707-829-0104 (fax)

We have a web page for this book, where we list errata, examples, and any additional information. You can access this page at:

http://oreilly.com/catalog/9781449388348/

Readers can request general help from the author and publisher through GetSatisfaction at:

http://getsatisfaction.com/oreilly

Readers may also file tickets for the sample code—as well as anything else in the book—through GitHub's issue tracker at:

http://github.com/ptwobrussell/Mining-the-Social-Web/issues

To comment or ask technical questions about this book, send email to:

bookquestions@oreilly.com

For more information about our books, conferences, Resource Centers, and the O'Reilly Network, see our website at:

http://www.oreilly.com

Acknowledgments

[XXX: Update this section once the 2e is ready for production]

To say the least, writing a technical book takes a *ridiculous* amount of sacrifice. On the home front, I gave up more time with my wife, Baseeret, and daughter, Lindsay Belle,

than I'm proud to admit. Thanks most of all to both of you for loving me in spite of my ambitions to somehow take over the world one day. (It's just a phase, and I'm really trying to grow out of it—honest.)

I sincerely believe that the sum of your decisions gets you to where you are in life (especially professional life), but nobody could ever complete the journey alone, and it's an honor give credit where credit is due. I am truly blessed to have been in the company of some of the brightest people in the world while working on this book, including a technical editor as smart as Mike Loukides, a production staff as talented as the folks at O'Reilly, and an overwhelming battery of eager reviewers as amazing as everyone who helped me to complete this book. I especially want to thank Abe Music, Pete Warden, Tantek Celik, J. Chris Anderson, Salvatore Sanfilippo, Robert Newson, DJ Patil, Chimezie Ogbuji, Tim Golden, Brian Curtin, Raffi Krikorian, Jeff Hammerbacher, Nick Ducoff, and Cameron Marlowe for reviewing material or making particularly helpful comments that absolutely shaped its outcome for the best. I'd also like to thank Tim O'Reilly for graciously allowing me to put some of his Twitter and Google+ data under the microscope; it definitely made those chapters much more interesting to read than they otherwise would have been. It would be impossible to recount all of the other folks who have directly or indirectly shaped my life or the outcome of this book.

Finally, thanks to you for giving this book a chance. If you're reading this, you're at least thinking about picking up a copy. If you do, you're probably going to find something wrong with it despite my best efforts; however, I really do believe that, in spite of the few inevitable glitches, you'll find it an enjoyable way to spend a few evenings/weekends and you'll manage to learn a few things somewhere along the line.

PART I
A Guided Tour of the Social Web

XXX: Add primer with overview of this part

CHAPTER 1

Mining the Twitterverse: Exploring Trending Topics, Discovering What People Are Talking About, and More

This chapter kicks off our journey of mining the social web with Twitter, a rich source of social data that needs no introduction. Twitter is a great starting point for social web mining, because of its inherent openness, clean and well documented API, rich developer tooling, and broad appeal to masses of users from every walk of life. In this chapter, we'll ease into the process of getting situated with a minimal (but effective) development environment with Python, survey Twitter's API, and distill some analytical insights from tweets. More specifically, we'll explore trending topics, narrow in on and collect tweets from a particular area of interest, distill some insights and, finally, visualize some of the initial analytical findings. Like most other chapters in this book, each chapter can do little more than scratch the surface of the broad possibilities. However, given that Twitter data is so accessible and open to public scrutiny, several additional chapters in Part II further elaborate on the rather broad swath of possibilities as a sort of case study in what's possible with social web data by using Twitter as a prototype. Let's get started mining the Twitterverse.

As you hopefully already know from reading the preface, the example code for this book is written with Python, and extra care has been taken to provide you with a virtual machine that provides a convenient turnkey development environment that should mitigate installation and configuration pains that sometimes occur. Although you might already have a recent version of Python on your system, please take special care to review the recommendations and "warm up" exercises outlined in (to come), which are intended for your benefit and maximum enjoyment of this book.

The Twitter Developer Platform

Twitter might be described as a real-time, highly social microblogging service that allows users to post short status updates called tweets that appear on timelines. Tweets may include one or more entities in its 140 characters of content and reference one or more places that map to a location in the real world. An understanding of users, tweets, entities, places, and timelines is essential to effective use of Twitter's API, so a brief introduction to Twitter Platform objects is in order before interacting with the API to fetch some data.

The first thing you probably already know but have not fully pondered yet is that a Twitter user need not be a real person; it could be a person, but it could also be a bot, a company, a musical group, an imaginary persona that only exists in someone's imagination, an impersonation of someone (living or dead), or just about anything else. There is very little governance of what a Twitter account could be aside from the badges you might see on some accounts that identify celebrities and public figures as "verified accounts" and basic restrictions in Twitter's Terms of Service agreement that is required for using the service. It may seem like a subtle distinction to mention, but it's an important distinction from some social web properties in which accounts must correspond to real living people, business entities, or an entity of a similar nature.

Next, unlike some social web properties such as Facebook and LinkedIn where a connection is bidirectional, Twitter's network of users is based upon an asymmetric model of following other users. In other words, you are either following, getting followed, or both in the case that both users are following one another. The interesting thing about the asymmetric nature of Twitter is that it allows you to keep up with the latest happenings of any other user, even though that other user may not choose to follow you back or even notice that you've followed them. This model is quite a contrast to a very symmetric social network like Facebook in which friendship requests must be approved and are therefore mutual.

Although caution has been taken in the preceding paragraph to introduce Twitter in terms of "following" relationships, the act of following someone could (and often is) equivalently described as the act of friending them, albeit it a special kind of "one-way" friendship. Thus, you'll often see user networks expressed as friends and followers even though the term friendship might be a misnomer in many cases. For example, if you were to follow Tim O'Reilly (@timoreilly) on Twitter, it would be accurate to describe the @timoreilly user as one of your friends on Twitter even though he may not know that you even exist online or in real life. However you choose to describe the network of Twitter users, just keep the distinction in mind and realize that you'll run across both descriptions in the API documentation and elsewhere.

Tweets are themselves the essence of Twitter, and while tweets are notionally thought of as the 140 characters of text content associated with a user's status update, there's really quite a bit more metadata there than meets the eye. In addition to the textual content of a tweet itself, tweets come bundled with two additional pieces of metadata that are of particular note: entities and places. Tweet entities are essentially the user mentions, hashtags, URLs, and media that may be associated with a tweet, and locations are places in the real world that may be attached to a tweet. Note that places may be the actual location in which a tweet was authored but it might also be a reference to the place that is described in a tweet.

To make it all a bit more concrete, let's consider a sample tweet with the following text:

"@ptwobrussell is writing @SocialWebMining, 2nd Ed. from his home office in Franklin, TN. Be #social: http://on.fb.me/16WJAf9"

The tweet is 124 characters long and contains four tweet entities: the user mentions @ptwobrussell and @SocialWebMining, the hashtag #social, and the URL http://on.fb.me/16WJAf9. Although there is a place called Franklin, TN that's explicitly mentioned in the tweet, the places metadata associated with the tweet might include the location in which the tweet was authored, which may or may not be Franklin, TN.

Finally, timelines are the chronologically-sorted natural habitats for tweets. Abstractly, you might say that a timeline is any particular collection of tweets that are meaningful curated and displayed in chronological order; however, it's a bit more common to see a few primary kinds of timelines with a couple of them being of particular note. From the perspective of an arbitrary Twitter user, the home timeline is the view that you see when you login to your account and look at all of the tweets from users that you are following whereas a particular user timeline is a collection of tweets only from a particular user. For example, when you log into your Twitter account, your home timeline that's filled with the tweets of those you are following is located at https://twitter.com whereas the URL for any particular user timeline must be suffixed with a context that identifies the user such as https://twitter.com/SocialWebMining. If you're interested in seeing what a particular user's home timeline looks like from their own perspective, you can access it with the additional following suffix appended to the URL. For example, what Tim O'Reilly sees on his home timeline when he logs into Twitter is accessible at https://twitter.com/timoreilly/following.

The public timeline is another important timeline that's quite interesting because it's literally the firehose of all tweets that has been known to peak out at hundreds of thousands of tweets per minute during events with particularly wide interest such as presidential debates. Even during particularly "quiet" times of day, Twitter's public timeline emits far too much data to consider for the scope of this book and presents interesting engineering challenges, which is at least one of the reasons that various third-party commercial vendors have partnered with Twitter to bring the firehose to the masses in a more drinkable fashion. All things considered, however, a small random sample of

the public timeline is available to API developers as well as a public streams API that provides filterable access to enough public data to engage in some interesting exercises.

The remainder of this chapter and Part II of this book assumes that you have a Twitter account, which is required for API access, so take a moment to crate one and, then, review Twitter's liberal terms of service, API documentation, and API Rules of the Road. The sample code for this chapter and Part II of the book generally doesn't require you to have any friends or followers of your own, but some of the examples in Part II of the book will be a lot more interesting and fun if you have an active account with a handful of friends and followers that you can use as a basis of social web mining. If you don't have an active account, now would be a good time to get plugged-in and start priming your account for the data mining fun to come.

We'll opt to make programmatic API requests with the Python, because the `twitter` package so elegantly mimics the RESTful API. If you're interested in seeing the raw requests that you could make with HTTP or exploring the API in a more interactive manner, however, check out the developer console or Twurl, a command line tool for doing the same.

Making Twitter API Requests

Twitter has taken great care to craft an elegantly-simple RESTful API that is intuitive and easy to use. Even so, there are great libraries available to mitigate the work involved in making API requests even further. A particularly beautiful Python package that wraps the Twitter API and mimics the public API semantics almost one-to-one is a package called `twitter`. Like most other Python packages, you can install it with `pip` by typing `pip install twitter` in a terminal.

Python Tip: Harnessing pydoc for Effective Help During Development

We'll work though some examples that illustrate the use of this package, but just in case you're ever in a situation where you need some help (and you will be), it's worth remembering that you can always skim the documentation for a package a few different ways. Outside of a Python shell, running `pydoc` on a package in your `PYTHONPATH` is a nice option. For example, `pydoc twitter` provides the package level documentation whereas `pydoc twitter.Twitter` provides documentation on the `Twitter` class. If you find yourself reviewing the documentation for certain modules often, you can elect to pass the `-w` option to `pydoc` and write out an HTML page that you can save and bookmark in your browser.

However, more than likely, you'll be in the middle of a working session when you need some help. The built-in `help()` function accepts a package or class name and is useful for an ordinary Python shell, whereas iPython users can suffix a package or class name

with a question mark to view inline help. It is highly recommended that you adopt iPython as your standard Python shell because of the various convenience functions such as tab completion, session history and "magic functions" that it offers. Recall that (to come) provides minimal details on getting oriented with recommended developer tools such as iPython.

Before you'll be able to make any API requests to Twitter, you'll need to create an application at https://dev.twitter.com/apps. Creating an application is the standard way for developers to gain API access and for Twitter to monitor and inteact with 3rd party platform developers as needed. The process for creating an application is pretty standard, and all that's needed is read-only access to the API. In the present context, you are creating an app that you are going to authorize to access your account data, so this might seem a bit roundabout; why not just plug in your username and password to access the API? While that approach might work fine for you, a third-party such as a friend or colleague probably wouldn't feel comfortable forking over a username/password combination in order to enjoy the same insights from your app. Giving up credentials is never a sound practice.

Fortunately, some smart people recognized this problem years ago, and now there's a standardized protocol called OAuth that works for these kinds of situations in a generalized way for the broader social web. The protocol is a social web standard at this point and stands for Open Authorization. If you remember nothing else from this tangent, just remember that OAuth is a means of allowing users to authorize 3rd party applications to access their account data without needing to share sensitive information like a password. (to come) provides a slightly broader overview of how OAuth works if you're more interested, and Twitter's specific OAuth documentation with some specific details about their particular implementation is also available.

Although it's an implementation detail, it may be worth nothing that Twitter's v1.1 API still implements OAuth 1.0A whereas many other social web properties have since upgraded to OAuth 2.0.

For simplicity of development, the key pieces of information that you'll need to take away from your newly created application's settings are its consumer key, consumer secret, access token, and access token secret. In tandem, these four credentials provide everything that an application would ultimately be getting to authorize itself through a series of redirects involving the user granting authorization, so treat them with the same sensitivity that you would a password. Figure 1-1 shows illustrates the context of retreiving these credentials.

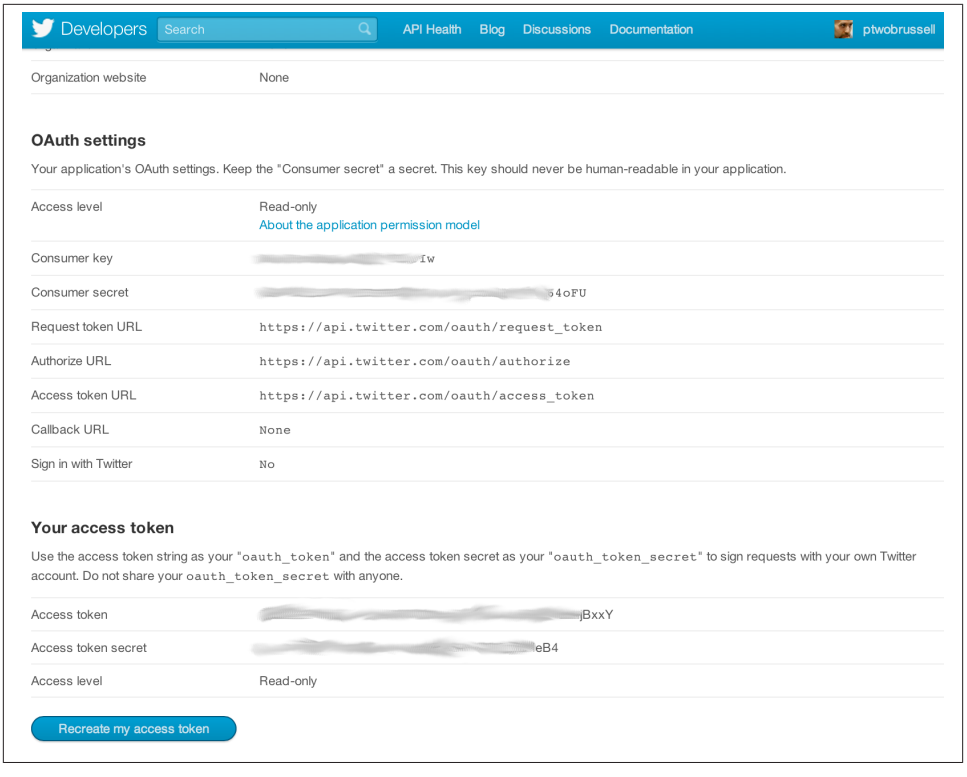


Figure 1-1. Create a new Twitter application to get OAuth credentials and API access

Without further ado, let's create an authenticated connection to Twitter's API and find out what people are talking about by inspecting the trends available to us through the GET trends/place resource. While you're at it, go ahead and bookmark the official API documentation as well as the REST API v1.1 Resources, because you'll be referencing them on a regular basis as you learn the ropes of the developer facing side of the Twitterverse.

As of March 2013, Twitter's API is now operates at version 1.1 and is significantly different in a few areas than the previous v1 API that you may have previously encountered. Version 1 of the API passed through a deprecation cycle of approximately six months and is no longer operational. All sample code in this book presumes version 1.1 of the API.

Let's fire up the Python interpreter and initiate a search. Try following along with Example 1-1, using Python's built-in help system as needed to try to answer as many of your own questions as possible before proceeding. While it's recommended that you go

through the execise of interactively following along in an interpreter, the source code for the remainder of the chapter is available in an iPython Notebook format and may be a more enjoyable way of following along.

Example 1-1. Authorizing an application to access Twitter account data

```
import twitter

# Go to http://twitter.com/apps/new to create an app and get these items
# See https://dev.twitter.com/docs/auth/oauth for more information
# on Twitter's OAuth implementation

CONSUMER_KEY = ''
CONSUMER_SECRET = ''
OAUTH_TOKEN = ''
OAUTH_TOKEN_SECRET = ''

auth = twitter.oauth.OAuth(OAUTH_TOKEN, OAUTH_TOKEN_SECRET,
                           CONSUMER_KEY, CONSUMER_SECRET)

twitter_api = twitter.Twitter(auth=auth)
```

Exploring Trending Topics

With an authorized API connection in place, you can now issue a request. Example 1-2 demonstrates how to ask Twitter for the topics that are currently trending worldwide, but keep in mind that the API can easily be parameterized to constrain the topics to more specific locales if you feel inclined to try out some of the possibilities. The device for constraining queries is via Yahoo! GeoPlanet's Where On Earth (WOE) ID system, which is an API unto itself that aims to provide a way to map a unique identifier to any named place on earth (or theoretically, even in a virtual world.) If you haven't already, go ahead and try out the example that collects a set of trends for both the entire world as well as the United States. You should be able to successfully see a semi-readable response from the API (as opposed to any kind of error message) before proceeding further.

Example 1-2. Retrieving trends

```
# The Yahoo! Where On Earth ID for the entire world is 1
# See https://dev.twitter.com/docs/api/1.1/get/trends/place and
# http://developer.yahoo.com/geo/geoplanet/

WORLD_WOE_ID = 1
US_WOE_ID = 23424977

# Prefix id with the underscore for query string parameterization.
# Without the underscore, the twitter package appends the id value
# to the URL itself as a special case keyword argument
```

```
world_trends = twitter_api.trends.place(_id=WORLD_WOE_ID)
us_trends = twitter_api.trends.place(_id=US_WOE_ID)
print world_trends
print
print us_trends
```

The pattern for using the `twitter` module is simple and predictable: instantiate the `Twitter` class with a base URL and then invoke methods on the object that correspond to URL contexts. For example, `twitter_api._trends.place(WORLD_WOE_ID)` initiates an HTTP call to GET https://api.twitter.com/1.1/trends/place.json?id=1. Note the URL mapping to the object chain that's constructed with the `twitter` package to make the request and how query string parameters are passed in as keyword arguments. In order to use the `twitter` package for arbitrary API requests, you generally just construct the request in that kind of straightforward manner with just a couple of minor caveats that we'll encounter soon enough.

It's worth noting that the developer documentation states that the results of this query are only updated once every 5 minutes, so it's not a judicious use of your efforts or API requests to ask for results more often than that frequency. Speaking of which, did you realize that Twitter imposes rate limits how many requests that an application can make to any given API resource within a given time window? Twitter's rate limits are well documented and each individual API resource also states its particular limits for your convenience. For example, the API request that we just issued for trends limits applications to 15 requests per 15 minute window. For more nuanced information on how Twitter's rate limits work, see REST API Rate Limiting in v1.1. For the purposes of following along in this chapter, it's highly unlikely that you'll get rate limited, and future chapters involving Twitter data will introduce some coping mechanisms.

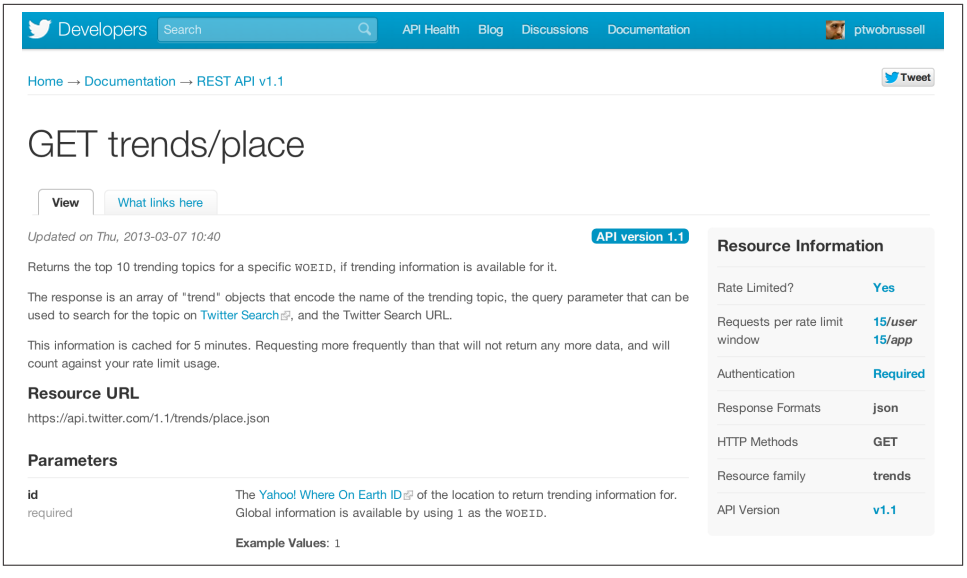


Figure 1-2. Rate limits for Twitter API resources are identified in the online documentation

Although it hasn't explicitly been stated yet, the semi-readable output from Example 1-2 is printed out as native Python data structures. Although an iPython interpreter will "pretty print" the output for you automatically, iPython Notebook and a standard Python interpreter will not. If you find yourself in these circumstances, you may find that it's useful to use the built-in `json` package to force a nicer display as illustrated in Example 1-3. If you're not yet familiar with JSON, it's a data exchange format that you will encounter on a regular basis. In a nutshell, JSON provides a way to arbitrarily store maps, lists, primimtives such as numbers and strings, and combinations thereof. In other words, you can theoretically model just about anything with JSON should you desire to do so.

Example 1-3. Displaying API responses as pretty printed JSON

```
import json
print json.dumps(world_trends, indent=1)
print
print json.dumps(us_trends, indent=1)
```

Example 1-4. An (abbreviated) sample response from the trends API

```
[
 {
  "created_at": "2013-03-27T11:50:40Z",
  "trends": [
   {
    "url": "http://twitter.com/search?q=%23MentionSomeoneImportantForYou",
```

```
    "query": "%23MentionSomeoneImportantForYou",
    "name": "#MentionSomeoneImportantForYou",
    "promoted_content": null,
    "events": null
   },
   ...
  ]
 }
]
```

Although it's easy enough to skim the two sets of trends and look for commonality, let's use Python's `sets` module to automatically compute this for us, because it's generally interesting to be able to take two arbitrary sets of things and compute on them. Example 1-5 demonstrates how to use a Python list comprehension to parse out the names of the trending topics from the results that were previously queried, cast those lists to sets, and take the intersection to reveal the common items between the sets.

Example 1-5. Computing the intersection of two sets of trends

```
world_trends_set = set([trend['name'] for trend in world_trends[0]['trends']])
us_trends_set = set([trend['name'] for trend in us_trends[0]['trends']])
common_trends = world_trends_set.intersection(us_trends_set)
print common_trends
```

Searching for Tweets

One of the common items between the sets of trending topics turns out to be the hashtag #MentionSomeoneImportantToYou, so let's use it as the basis of a search query to fetch some tweets for further analysis. Example 1-6 illustrates how to exercise the GET search/tweets resource for a particular query of interest, including the ability to use a special field that's included in the metadata for the search results to easily make additional requests for more search results.

Coverage of Twitter's Streaming API resources is out of scope for this chapter but may be more appropriate for many situations in which you want to maintain a constantly updated view of tweets.

Example 1-6. Collecting search results

```
# Pick a trending topic, or anything else for that matter
q = '#MentionSomeoneImportantForYou'

count = 100

# See https://dev.twitter.com/docs/api/1.1/get/search/tweets

search_results = twitter_api.search.tweets(q=q, count=count)
```

```
statuses = search_results['statuses']

# Iterate through 5 more batches of results by following the cursor

for _ in range(5):
    try:
        next_results = search_results['search_metadata']['next_results']
    except KeyError, e: # No more results when next_results doesn't exist
        break

    # Create a dictionary from next_results, which has the following form:
    # ?max_id=313519052523986943&q=%23MentionSomeoneImportantForYou&include_entities=1
    kwargs = dict([ kv.split('=') for kv in next_results[1:].split("&") ])

    search_results = twitter_api.search.tweets(**kwargs)
    statuses += search_results['statuses']
```

Although we're just passing in a hashtag to the search API at this point, it's well worth noting that it contains a number of powerful operators that allow you to filter queries according to the existence or non-existence of various keywords, originator of the tweet, location associated with the tweet, etc.

In essence, all the code does is repeatedly make requests to the search API. One thing that might initially catch you off guard if you've worked with other web APIs (including version 1 of Twitter's API) is that there's no explicit concept of pagination in the search API itself. Reviewing the API documentation reveals that this is a very intentional decision, and there are some good reasons for taking a cursoring approach instead given the highly dynamic state of Twitter resources. The best practices for cursoring varies a bit throughout the Twitter developer platform with the search API providing a slightly simpler way of navigating search results than other resources such as timelines. Search results contain a special `search_metadata` field that includes a `next_results` with a query string that provides the basis of a subsequent query. If we weren't using a library like `twitter` to facilitate making the HTTP requests for us, this preconstructed query string would just be appended to the API search URL, and we'd update it with additional parameters for handling OAuth. However, since we are not making our HTTP requests directly, we must parse the query string into its constituent key-value pairs and provide them as keyword arguments.

In Python parlance, we are unpacking the values in a dictionary into keyword arguments that the function receives. In other words, the function call inside of the `for` loop in Example 1-6 ultimately invokes a function such as`twitter_api.search.tweets(q='%23MentionSomeoneImportantForYou', include_entities=1, max_id=313519052523986943)` even though it appears in the source code as

`twitter_api.search.tweets(**kwargs)` with `kwargs` being a dictionary of key-value pairs.

The `search_metadata` field also contains a `refresh_url` value that can be used if you'd like to maintain and periodically update your collection of results with new information that's become available since the previous query.

Example 1-7 displays a tweet from the search results for #MentionSomeoneImportantForYou. Take a moment to peruse (all of) it. As was alluded to earlier, there's a lot more to a tweet than meets the eye. the 140 characters of possible content. The particular tweet below is fairly representative and in excess of 5KB of total content when represented in uncompressed JSON. That's more than 40 times the amount of data that the 140 characters of text that's normally thought of as a tweet.

Example 1-7. A sample tweet from search results

```
[
 {
  "contributors": null,
  "truncated": false,
  "text": "RT @hassanmusician: #MentionSomeoneImportantForYou God.",
  "in_reply_to_status_id": null,
  "id": 316948241264549888,
  "favorite_count": 0,
  "source": "<a href=\"http://twitter.com/download/android\"...",
  "retweeted": false,
  "coordinates": null,
  "entities": {
   "user_mentions": [
    {
     "id": 56259379,
     "indices": [
      3,
      18
     ],
     "id_str": "56259379",
     "screen_name": "hassanmusician",
     "name": "Download the NEW LP!"
    }
   ],
   "hashtags": [
    {
     "indices": [
      20,
      50
     ],
     "text": "MentionSomeoneImportantForYou"
```

```
  }
 ],
 "urls": []
},
"in_reply_to_screen_name": null,
"in_reply_to_user_id": null,
"retweet_count": 23,
"id_str": "316948241264549888",
"favorited": false,
"retweeted_status": {
 "contributors": null,
 "truncated": false,
 "text": "#MentionSomeoneImportantForYou God.",
 "in_reply_to_status_id": null,
 "id": 316944833233186816,
 "favorite_count": 0,
 "source": "web",
 "retweeted": false,
 "coordinates": null,
 "entities": {
  "user_mentions": [],
  "hashtags": [
   {
    "indices": [
     0,
     30
    ],
    "text": "MentionSomeoneImportantForYou"
   }
  ],
  "urls": []
 },
 "in_reply_to_screen_name": null,
 "in_reply_to_user_id": null,
 "retweet_count": 23,
 "id_str": "316944833233186816",
 "favorited": false,
 "user": {
  "follow_request_sent": null,
  "profile_use_background_image": true,
  "default_profile_image": false,
  "id": 56259379,
  "verified": false,
  "profile_text_color": "3C3940",
  "profile_image_url_https": "https://si0.t...",
  "profile_sidebar_fill_color": "95E8EC",
  "entities": {
   "url": {
    "urls": [
     {
      "url": "http://t.co/yRX89YM4J0",
      "indices": [
```

```
      0,
      22
     ],
     "expanded_url": "http://www.datpiff....",
     "display_url": "datpiff.com/mixtapes-detai\u2026"
    }
   ]
  },
  "description": {
   "urls": []
  }
 },
 "followers_count": 105041,
 "profile_sidebar_border_color": "000000",
 "id_str": "56259379",
 "profile_background_color": "000000",
 "listed_count": 64,
 "profile_background_image_url_https": "https://si0.t...",
 "utc_offset": -18000,
 "statuses_count": 16691,
 "description": "#TheseAreTheWordsISaid LP",
 "friends_count": 59615,
 "location": "",
 "profile_link_color": "91785A",
 "profile_image_url": "http://a0.twimg.com/...",
 "following": null,
 "geo_enabled": true,
 "profile_banner_url": "https://si0.twimg.com/pr...",
 "profile_background_image_url": "http://a0.twi...",
 "screen_name": "hassanmusician",
 "lang": "en",
 "profile_background_tile": false,
 "favourites_count": 6142,
 "name": "Download the NEW LP!",
 "notifications": null,
 "url": "http://t.co/yRX89YM4J0",
 "created_at": "Mon Jul 13 02:18:25 +0000 2009",
 "contributors_enabled": false,
 "time_zone": "Eastern Time (US & Canada)",
 "protected": false,
 "default_profile": false,
 "is_translator": false
},
"geo": null,
"in_reply_to_user_id_str": null,
"lang": "en",
"created_at": "Wed Mar 27 16:08:31 +0000 2013",
"in_reply_to_status_id_str": null,
"place": null,
"metadata": {
 "iso_language_code": "en",
 "result_type": "recent"
```

```
 }
},
"user": {
 "follow_request_sent": null,
 "profile_use_background_image": true,
 "default_profile_image": false,
 "id": 549413966,
 "verified": false,
 "profile_text_color": "3D1957",
 "profile_image_url_https": "https://si0.twimg....",
 "profile_sidebar_fill_color": "7AC3EE",
 "entities": {
  "description": {
   "urls": []
  }
 },
 "followers_count": 110,
 "profile_sidebar_border_color": "FFFFFF",
 "id_str": "549413966",
 "profile_background_color": "642D8B",
 "listed_count": 1,
 "profile_background_image_url_https": "https:...",
 "utc_offset": 0,
 "statuses_count": 1294,
 "description": "i BELIEVE do you? I admire n adore @justinbieber ",
 "friends_count": 346,
 "location": "All Around The World ",
 "profile_link_color": "FF0000",
 "profile_image_url": "http://a0.twimg.com/pr...",
 "following": null,
 "geo_enabled": true,
 "profile_banner_url": "https://si0.twimg.com/...",
 "profile_background_image_url": "http://a0.tw...",
 "screen_name": "LilSalima",
 "lang": "en",
 "profile_background_tile": true,
 "favourites_count": 229,
 "name": "KoKo :D",
 "notifications": null,
 "url": null,
 "created_at": "Mon Apr 09 17:51:36 +0000 2012",
 "contributors_enabled": false,
 "time_zone": "London",
 "protected": false,
 "default_profile": false,
 "is_translator": false
},
"geo": null,
"in_reply_to_user_id_str": null,
"lang": "en",
"created_at": "Wed Mar 27 16:22:03 +0000 2013",
"in_reply_to_status_id_str": null,
```

```
  "place": null,
  "metadata": {
   "iso_language_code": "en",
   "result_type": "recent"
  }
 },
 ...
]
```

Basic Anatomy of a Tweet

The online documentation is always the definitive source for Twitter platform objects, and its worthwhile bookmarking the Tweets page, because it's one that you'll refer to quite frequently as you get familiarized with the basic anatomy of a tweet. No attempt is made here or elsewhere to regurgitate online documentation, but a few notes are of interest given that you might still be a bit overwhelmed by the 5KB of information that comprises a tweet. For simplicity of nomenclature, let's assume that we've extracted a single tweet from the search results and stored it to a variable named `t`. For example, `t.keys()` returns the top level fields for the tweet and `t['id']` would access the identifier of the tweet.

If you're following along with the iPython Notebook for this chapter, the exact tweet that's under scrutiny is stored in a variable named `t` so that you can interactively access its fields and explore more easily. The current discussion assumes the same nomenclature, so values should correspond one-for-one.

- The human readable text of a tweet is available through `t['text']`

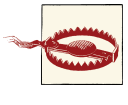
  ```
  RT @hassanmusician: #MentionSomeoneImportantForYou God.
  ```

- The entities in the text of a tweet are conveniently processed for you and available through `t['entities']`

  ```
  {
   "user_mentions": [
    {
     "indices": [
      3,
      18
     ],
     "screen_name": "hassanmusician",
     "id": 56259379,
     "name": "Download the NEW LP!",
     "id_str": "56259379"
    }
   ],
  ```

```
    "hashtags": [
     {
      "indices": [
       20,
       50
      ],
      "text": "MentionSomeoneImportantForYou"
     }
    ],
    "urls": []
   }
```

- Clues as to the "interestingness" of a tweet are available through `t['favorite_count']` and `t['retweet_count']`, the number of times it's been bookmarked or retweeted, respetively.
- If a tweet has been retweeted, the `t['retweeted_status']` field provides significant detail about the original tweet itself and its author. Keep in mind that sometimes the text of a tweet changes as it is retweeted by way of users adding reactions or otherwise manipulating the text.
- The `t['retweeted']` field denotes whether or not the authenticated user (via an authorized application) has retweeted this particular tweet. Fields that vary depending upon the point of view of the particular user are denoted in Twitter's developer documentation as perspectival, which means that its value will vary depending upon the perspective of the user.
- Additionally, note that only original tweets are retweeted from the standpoint of the API and information management. Thus, the `retweet_count` reflects the total number of times that the original tweet has been retweeted and should reflect the same value in both the original tweet and all subsequent retweets. In other words, retweets aren't retweeted. If you think you're retweeting a retweet, you're actually just retweeting the original tweet that you were exposed to through a proxy.



A common mistake is to check the value of the `retweeted` field to determine whether or not a tweet has ever been retweeted by anyone. To check whether a tweet has ever been retweeted, you should instead check that a `retweeted_status` node wrapper exists in the tweet.

You should tinker around with the sample tweet and consult the documentation to clarify any lingering questions you might have before moving forward. A good working knowledge of a tweet's anatomy is critical to effectively mining Twitter data.

Discovering What People Are Talking About

Let's distill the entities and the text of the tweets into a convenient data structure for further examination. Example 1-8 extracts the text, screen names, and hashtags from the tweets that are collected and introduces a Python idiom called a double list comprehension. If you understand a (single) list comprehension, the code formatting should serve to illustrate the intuition behind the double list comprehension being nothing more than a nested loop. List comprehensions are particularly powerful because they usually yield substantial performance gains over nested lists and provide an intuitive (once you're familiar with them) yet terse syntax. List comprehensions are used frequently throughout this book, and it's worth consulting the "Data Structures" section in the official Python tutorial for more details if you'd like additional context.

Example 1-8. Extracting text, screen names, and hashtags from tweets

```
status_texts = [ status['text'] for status in statuses ]

screen_names = [ user_mention['screen_name']
                 for status in statuses
                     for user_mention in status['entities']['user_mentions'] ]

hashtags = [ hashtag['text']
                 for status in statuses
                     for hashtag in status['entities']['hashtags'] ]
```

With some interesting data on hand, let's explore it. One of the more intuitive measurements that can be applied to unstructured text is a metric called lexical diversity. Mathematically, this is an expression of the number of unique tokens in the text divided by the total number of tokens in the text, which are both elementary yet important metrics in and of themselves. Lexical diversity is a concept that is interesting with respect to interpersonal communications because it provides a quantitative measure for the diversity of an individual's or group's vocabulary. For example, suppose you are listening to someone speaking who repeatedly says "and stuff" to broadly generalize information as opposed to providing specific examples to reinforce points with more detail or clarity. Now, contrast that speaker to someone else who seldom uses the word "stuff" to generalize and instead reinforces points with very specific concrete examples. The speaker who repeatedly says "and stuff" would have a lower lexical diversity than the speaker who uses a more diverse vocabulary, and chances are reasonably good that you'd walk away from the conversation feeling as though the speaker with the higher lexical diversity understands the subject matter better.

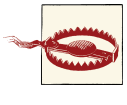
As applied to tweets or similar online communications, lexical diversity can be interesting to consider as a primitive statistic for answering a number of questions such as how broad or narrow the subject matter is that an individual or group discusses. Although an overall assessment could be interesting, breaking down the analysis to specific time periods could yield additional insight as could comparing different groups or in-

dividuals. For example, it would be interesting to measure whether or not there is a significant difference between the lexical diversity for two soft-drink companies such as Coca-Cola and Pepsi as an entry point for exploration if you were comparing the effectiveness of their social media marketing campaigns on Twitter. Virtually all analysis starts out with the simple exercise of counting things, and much of what we'll be doing in this book is manipulating data so that it can be counted and manipulated in meaningful ways. From an empirical standpoint, counting observable things is the starting point for just about everything and, thus, the initial starting point for any kind of statistical filtering or manipulation that serves to find what may be faint signal in noisy data.

With a basic understanding of how to use a statistic like lexical diversity to analyze textual content such as tweets, let's now compute the lexical diversity for statuses, screen names, and hashtags for our working data set as shown in Example 1-9.

Example 1-9. Calculating lexical diversity for tweets

```
# A function for computing lexical diversity
def lexical_diversity(tokens):
    return 1.0*len(set(tokens))/len(tokens)

# A function for computing the average number of words per tweet
def average_words(statuses):
    total_words = sum([ len(s.split()) for s in statuses ])
    return 1.0*total_words/len(statuses)

# Compute a collection of all words from all tweets
words = [ w
          for t in status_texts
              for w in t.split() ]

print lexical_diversity(words)         # 0.67610619469
print lexical_diversity(screen_names)  # 0.955414012739
print lexical_diversity(hashtags)      # 0.0686274509804
print average_words(status_texts)      # 5.76530612245
```



Prior to Python 3.0, the division operator applies the floor function and returns an integer value (unless one of the operands is a floating-point value). Multiply either the numerator or the denominator by 1.0 to avoid truncation errors.

In analyzing the results, there are a few observations worth considering:

- The lexical diversity for the words in the text of the tweets is around 0.67, and one way to interpret a lexical diversity of around 0.67 would be to say that about two out of every three words is unique; another expression of the same interpretation might be that each status update carries around 67% unique information. Given that the average number of words in each tweet is around 6, that translates to around

four unique words per tweet. Intuition aligns with the data in that the nature of a #MentionSomeoneImportantForYou trending hashtag is to solicit a response of what will probably be a few words long. In any event, a value of 0.67 is on the high side for lexical diversity of ordinary human communication, but given the nature of the data, it seems very reasonable.

- A lexical diversity value for the screen names, however, is even higher with a value of 0.95, which means that about 19 out of 20 screen names mentioned are unique. This observation also makes sense given that many answers to the question will be a screen name and that most people won't be providing the same responses for the solicitous hashtag.
- The lexical diversity of hashtags is extremely low at a value of around 0.068 implying that very few values other than the #MentionSomeoneImportantForYou hashtag appear in results. Again, this makes good sense given that the most responses are short and that hashtags really wouldn't make much sense to introduce as a response for to the question of mentioning someone important for you.
- The average number of words per tweet is very low at a value of just under 6, which makes sense given the nature of the hashtag that is designed to solicit short responses consisting of just a few words.

What would be interesting at this point is to zoom in on some of the data and see if there were any common responses to the query or other insights that could come from a more qualitative analysis. Given an average number of words per tweet as low as 6, it's unlikely that users applied any abbreviations to stay within the 140 characters, so the amount of noise for the data should be remarkably low, and additional frequency analysis may reveal some interesting things. As of Python 2.7, a `collections` module is available that provides a counter that makes computing a frequency distribution rather trivial. Example 1-10 demonstrates how to use a `Counter` to compute frequency distributions as ranked lists of terms.

Among the more compelling reasons for mining Twitter data is to try to answer the question of what people are talking about *right now*. One of the simplest techniques you could apply to answer this question is basic frequency analysis just as we are performing here.

Example 1-10. Creating a basic frequency distribution from the words in tweets

```
from collections import Counter

for item in [words, screen_names, hashtags]:
    c = Counter(item)
    print c.most_common()[:10] # top 10
    print
```

Example 1-11. Sample results from frequency analysis of tweets

```
[(u'#MentionSomeoneImportantForYou', 92), (u'RT', 34), (u'my', 10),
 (u',', 6), (u'@justinbieber', 6), (u'<3', 6), (u'My', 5), (u'and', 4),
 (u'I', 4), (u'te', 3)]

[(u'justinbieber', 6), (u'Kid_Charliej', 2), (u'Cavillafuerte', 2),
 (u'touchmestyles_', 1), (u'aliceorr96', 1), (u'gymleeam', 1), (u'fienas', 1),
 (u'nayely_1D', 1), (u'angelchute', 1), (u'BigTittieAngel', 1)]

[(u'MentionSomeoneImportantForYou', 94), (u'mentionsomeoneimportantforyou', 3),
 (u'NoHomo', 1), (u'Love', 1), (u'MentionSomeOneImportantForYou', 1), (u'MyHeart', 1),
 (u'bebesito', 1)]
```

The results the frequency distribution is a map of key-value pairs corresponding to terms and frequencies for those terms, so let's make reviewing the results a little easier on the eyes by emitting a tabular format. A package called `prettytable` can be installed by typing `pip install prettytable` in a terminal and provides a convenient way to emit a fixed width format, and Example 1-12 shows how to use it to display the same results.

Example 1-12. Using prettytable to display tuples in a nice tabular format

```
from prettytable import PrettyTable

for label, data in (('Word', words),
                    ('Screen Name', screen_names),
                    ('Hashtag', hashtags)):
    pt = PrettyTable(field_names=[label, 'Count'])
    c = Counter(words)
    [ pt.add_row(kv) for kv in c.most_common()[:10] ]
    pt.align[label], pt.align['Count'] = 'l', 'r' # Set column alignment
    print pt
```

The results from Example 1-12 are displayed in Example 1-13. The result format is a series of nicely formatted text-based tables that are easy to skim.

Example 1-13. Frequency distributions displayed in a tabular format

```
+-------------------------------+-------+
| Word                          | Count |
+-------------------------------+-------+
| #MentionSomeoneImportantForYou |    92 |
| RT                            |    34 |
| my                            |    10 |
| ,                             |     6 |
| @justinbieber                 |     6 |
| &lt;3                         |     6 |
| My                            |     5 |
| and                           |     4 |
| I                             |     4 |
```

```
| te                              |     3 |
+---------------------------------+-------+
+----------------+-------+
| Screen Name    | Count |
+----------------+-------+
| justinbieber   |     6 |
| Kid_Charliej   |     2 |
| Cavillafuerte  |     2 |
| touchmestyles_ |     1 |
| aliceorr96     |     1 |
| gymleeam       |     1 |
| fienas         |     1 |
| nayely_1D      |     1 |
| angelchute     |     1 |
| BigTittieAngel |     1 |
+----------------+-------+
+-------------------------------+-------+
| Hashtag                       | Count |
+-------------------------------+-------+
| MentionSomeoneImportantForYou |    94 |
| mentionsomeoneimportantforyou |     3 |
| NoHomo                        |     1 |
| Love                          |     1 |
| MentionSomeOneImportantForYou |     1 |
| MyHeart                       |     1 |
| bebesito                      |     1 |
+-------------------------------+-------+
```

A very quick skim of the results from Example 1-13 reveals at least one marginally surprising thing: Justin Bieber is high on the list of entities for this small sample of data, and given his popularity with tweens on Twitter, may very well have been the "most important someone" for this trending topic, though the results here are inconclusive. Although the entities with a frequency greater than two are interesting, the broader results are also revealing in other ways. For example, "RT" was a very common token, implying that there were a significant number of retweets. Finally, as might be expected, the #MentionSomeoneImportantForYou hashtag and a couple of case-sensitive variations dominated the hashtags; a takeaway is that it would be worthwhile to normalize each word, screen name, and hashtag to lowercase when tabulating frequencies since there will inevitably be variation in tweets.

Even though the user interface and many Twitter clients have long since adopted the native retweet API that is used to populate status values such as `retweet_count` and `retweeted_status`, some Twitter users may still use older client applications or elect to add commentary to a tweet, which entails a now somewhat antiquated workflow involving a copy-and-paste of the text with a prepending of the convention "RT @user" to give proper credit. In cases such as these, `retweet_count` and `retweeted_status` will not reflect that the retweet metadata and detection and parsing of the "RT @user" or similar retweeting conventions is the best heuristic that you can use. See [XXX: add xref for recipe for this in Part II when ready] for more details.

A very intresting exercise at this point would be to further analyze the data to determine if there was a particular tweet that was highly retweeted or if there were just lots of "one-off" retweets. The approach we'll take to find the most popular retweets is to simply iterate over each status update and store out the retweet count, originator of the retweet, and text of the retweet if the status update is a retweet. Example 1-14 demonstrates how to capture these values with a list comprehension and sort by the retweet count to display the top few results.

Example 1-14. Finding the most popular retweets

```
retweets = [
            # Store out a tuple of these three values ...
            (status['retweet_count'],
             status['retweeted_status']['user']['screen_name'],
             status['text'])

            # ... for each status ...
            for status in statuses

            # ... so long as the status meets this condition.
                if status.has_key('retweeted_status')
           ]

# Slice off the top 5 from the sorted results and display each item in the tuple
for count, screen_name, text in sorted(retweets, reverse=True)[:5]:
    print count, screen_name, text
```

Results from Example 1-14 are interesting:

Example 1-15. Sample output from Example 1-14

```
+-------+---------------+-----------------------------------------------------+
| Count | Screen Name   | Text                                                |
+-------+---------------+-----------------------------------------------------+
| 23    | hassanmusician | RT @hassanmusician: #MentionSomeoneImportantForYou |
|       |               | God.                                                |
| 21    | HSweethearts  | RT @HSweethearts: #MentionSomeoneImportantForYou    |
```

```
|       |                | my high school sweetheart                          |
| 15    | LosAlejandro_  | RT @LosAlejandro_: ¿Nadie te menciono en           |
|       |                | "#MentionSomeoneImportantForYou"? JAJAJAJAJAJAJA   |
|       |                | JAJAJAJAJAJAJAJAJAJAJAJAJAJAJAJAJAJA Ven, ...     |
| 9     | SCOTTSUMME     | RT @SCOTTSUMME: #MentionSomeoneImportantForYou My  |
|       |                | Mum. Shes loving, caring, strong, all in one. I    |
|       |                | love her so much                                   |
| 7     | degrassihaha   | RT @degrassihaha: #MentionSomeoneImportantForYou I |
|       |                | can't put every Degrassi cast member, crew member, |
|       |                | and writer in just one tweet....                   |
+-------+----------------+----------------------------------------------------+
```

Interestingly, "God" tops the list, followed closely by "my high school sweetheart", and coming in at number four on the list is "My Mum." None of the top five items in the list corresond to Twitter user accounts, although we might have suspected this (with the exception of @justinbieber) from the previous analysis. Inspection of results further down the list do reveal particular user mentions, but the sample we have drawn from for this query is so small that no trends emerge. Searching for a larger sample of results would likely yield some user mentions with a frequency greater than one, which would be interesting to further analyze. The possibilities for further anlaysis are pretty open-ended, and by now, hopefully, you're itching to try out some custom queries of your own.

Before moving on, a subtlety worth realizing is that it's quite possible (and probable given the relatively low frequencies of the retweets observed in this section) is that the original tweets that were retweeted may not exist in our sample search results set. For example, the most popular retweet in the sample results originated from a user with a screen name of @hassanmusician and was retweeted 23 times. However, closer inspection of the data reveals that we only collected one of the 23 retweets in our search results. Neither the original tweet nor any of the other 22 retweets appear in the data set. This doesn't pose any particular problems, although it might beg the interesting question of who were the other 22 retweeters for this status. The answer to this kind of question is a very valuable one because it allows us to take content that represents a concept such as "God" in this case, and discover a group of other users who would apparently share the same sentiment or common interest. A handy way to model data involving people and things that they're interested in is called an interest graph, which is the primary data structure that supports analysis in [XXX: xref GitHub Chapter when it's added to the repo]. Interpretative speculation about these users could suggest that they are spiritual or religious individuals, and further analysis of their particular tweets might corroborate that kind of claim. Example 1-16 shows how to find these individuals with the GET statuses/retweets/:id API.

Example 1-16. Looking up users who have retweeted a status

```
# Get the original tweet id for a tweet from its retweeted_status node

_retweets = twitter_api.statuses.retweets(id=316944833233186816)
```

```
[r['user']['screen_name'] for r in _retweets]
```

Further analysis of the users who retweeted this particular status for any particular religious or spiritual affiliation is left as an exercise.

Visualizing Frequency Data with Histograms

A nice feature of IPython Notebook is its ability to generate and insert very high quality and customizable plots of data as part of interactive workflows. In particular, the `matplotlib` package and other scientific computing tools that are available for IPython Notebook are quite powerful and capable of generating complex figures with very little effort once you understand the basic workflows. To illustrate the use of `matplotlib`'s plotting capabilities, let's plot some data for display. To get warmed up, let us consider a plot that displays the frequencies for the words in Example 1-13. Given a sorted list of tuples where each tuple is a `(word, frequency)` pair, x-axis value will correspond to the index of the tuple, and the y-axis will correspond to frequency for the word in that tuple. It would generally be impractical to try and plot each word as a value on the x-axis, although that's what the x-axis is representing. Figure 1-3 displays a plot for the words given in Example 1-13. The plot can be generated directly in IPythonNotebook with the code shows in Example 1-17.

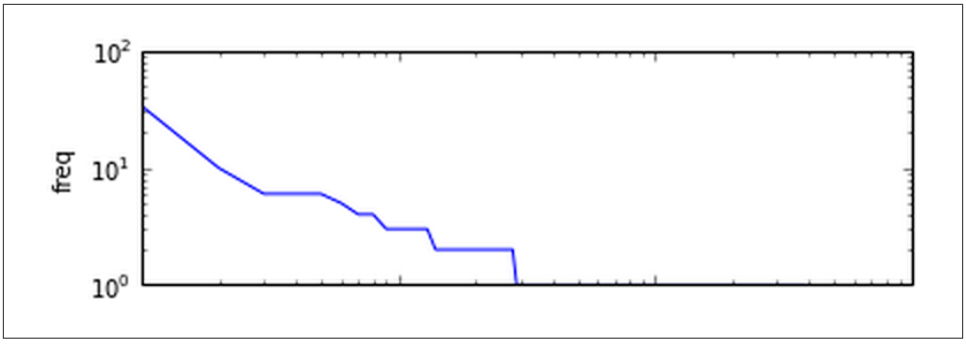


Figure 1-3. A plot displaying the sorted frequencies for each word from Example 1-13

Example 1-17. Plotting frequencies of words

```
import matplotlib.pylab as plt

word_counts = sorted(Counter(words).values(), reverse=True)

# Construct a figure and add a subplot and its data
fig = figure()
ax = fig.add_subplot(2,1,1) #2x1 grid, 1st subplot in the fig
ax.plot(word_counts)
```

```
# Format the plot
ax.set_xscale('log')
ax.set_yscale('log')
ax.set_xticklabels("")
ax.set_ylabel("freq")
```

A plot of frequency values is intuitive and convenient, but it can also be useful to group together data values into bins that correspond to a range of frequencies. For example, how many words have frequency between 1 and 5, between 5 and 10, between 10 and 15, and so forth? A histogram is designed for precisely this purpose and provides a convenient visualization for displaying tabulated frequencies as adjacent rectangles where the area of each rectange is a measure of the data values that fall within that particular range of values. Figure 1-4 and Figure 1-5 show histograms of the tabular data from Example 1-13 and Example 1-15, respectively. Although the histogram doesn't have x-axis labels that show us which words have which frequencies, that's not really its purpose. Its purpose is to give us insight into the underlying frequency distribution with the x-axis corresponding to a range for words that each have a frequency within that range and the y-axis corresponding to the total frequency of all words that appear within that range.

When interpreting Figure 1-4, look back to the corresponding tabular data and consider that there is a large number of words, screen names, or hashtags that have very low frequencies and appear very few times in the text; however, when you take all of these low frequency terms and bin them together into a range of "all words with frequency between one and ten", we see that the total number of these low frequency words accounts for most of the text. More concretely, we see that there are approximately ten words that account for almost all of the frequencies as rendered by the area of the large blue rectangle, while there are just a couple of words with much higher frequencies, and we know that these words are "#MentionSomeoneImportantForYou" and "RT" with corresonding frequencies of 34 and 92 as given by our tabulated data.

Likewise, when interpreting Figure 1-5, we see that there are a select few tweets that are retweeted with a much higher frequencies than the majority of the tweets, which are retweeted only once and account for the majority of the volume given by the largest blue rectangle on the left side of the histogram.

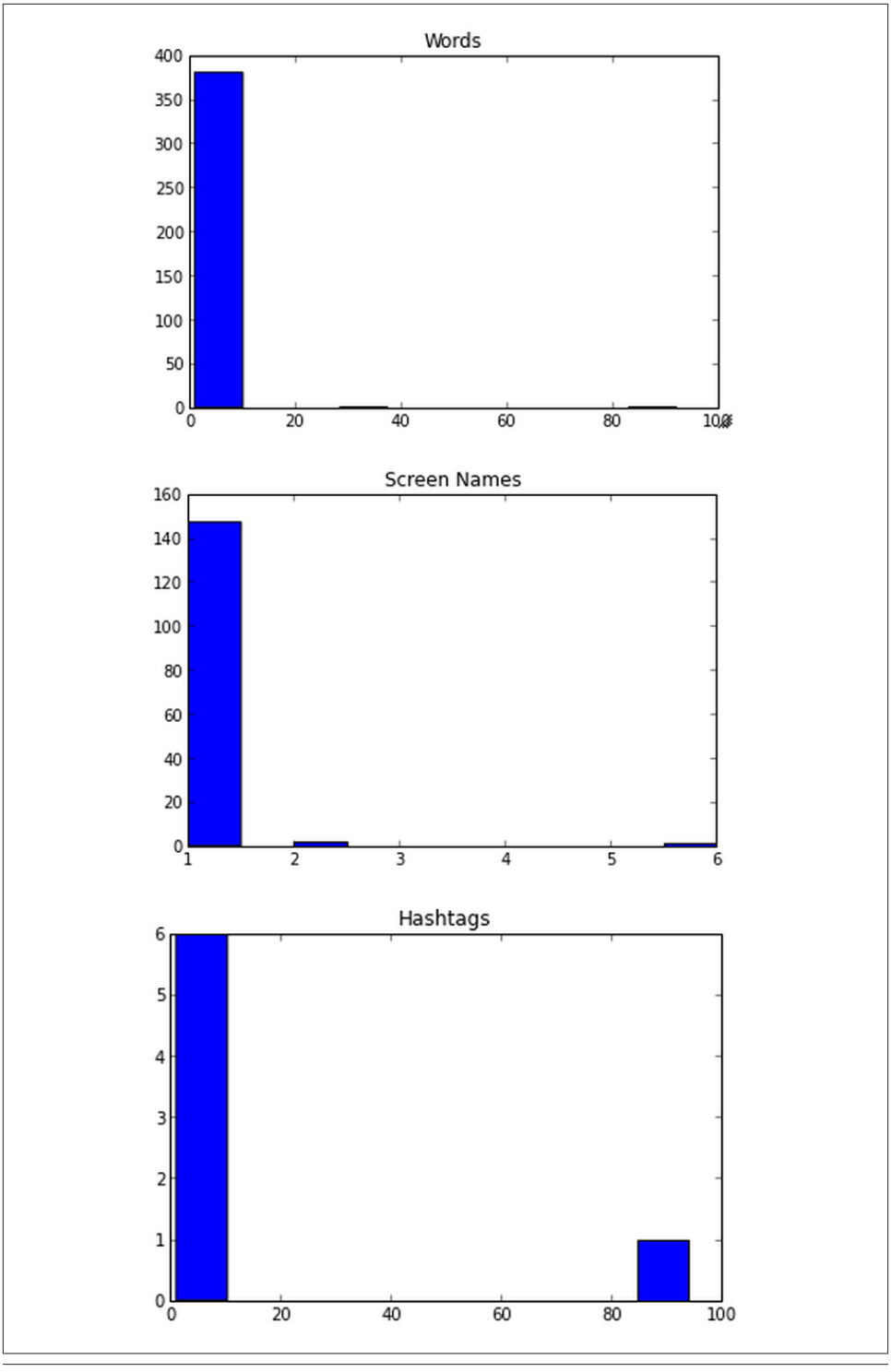


Figure 1-4. Histograms of tabulated frequency data for words, screen names, and hashtags

tags

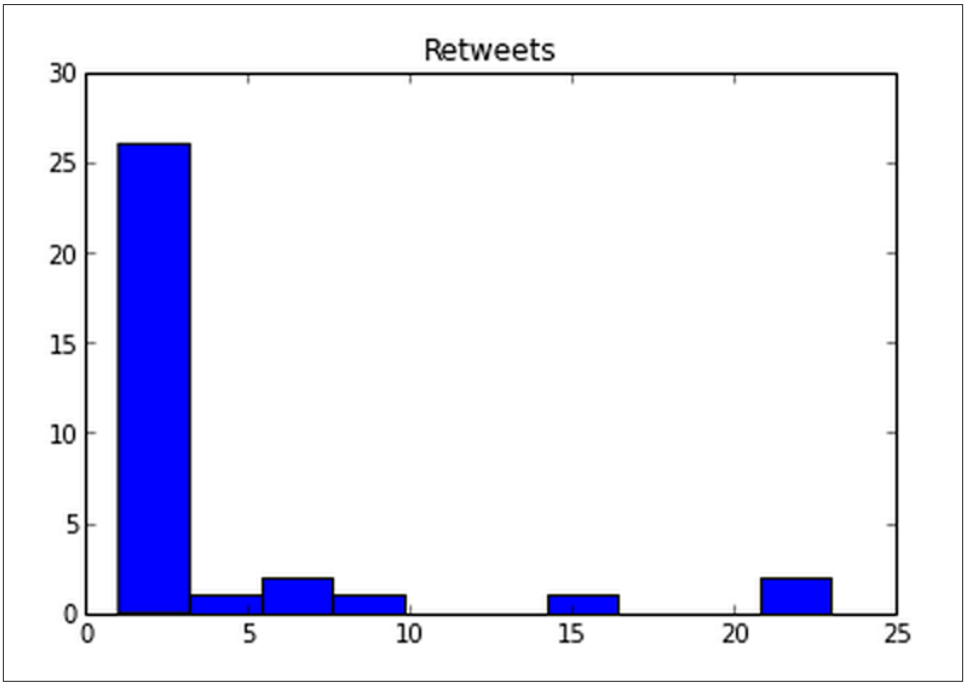
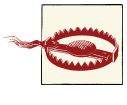


Figure 1-5. A histogram of retweet frequencies

The code for generating these histograms directly in IPython Notebook is given in Example 1-18 and Example 1-19. Taking some time to explore the capabilities of matplotlib and other scientific computing tools is a worthwhile investment.



Installation of scientific computing tools such as `matplotlib` can potentially be a frustrating experience because of certain dynamically loaded libraries in the dependenc chain, and the pain involved can vary from version to version and operating system to operating system. It is highly recommended that you use the virtual machine that is provided with this book at [XXX: xref with a ulink] if you don't already have these tools installed, don't feel confident that you won't burn up a lot of time installing them, or just want to take the easy way out. (After all, that's why the virutal machine was provided.) If you feel like being adventurous, you can try to `pip install matplotlib` and follow the dependency chain as you encounter errors along the way that require additional packages such as `numpy`, which can also be installed with `pip`.

Example 1-18. Generating histograms of words, screen names, and hashtags

```
for label, data in (('Words', words),
                    ('Screen Names', screen_names),
                    ('Hashtags', hashtags)):

    # Build a frequency map for each set of data
    # and plot the values
    c = Counter(data)
    plt.hist(c.values())

    # Add a title and display as a new figure
    plt.title(label)
    plt.figure()
```

Example 1-19. Generating a histogram of retweet counts

```
# Using underscores while unpacking values in
# a tuple is idiomatic for discarding them
counts = [count for count, _, _ in retweets]
plt.hist(counts)
print counts
plt.title("Retweets")
plt.figure()
```

Closing Remarks

This chapter got you up and running with Twitter's API, and illustrated how easy (and fun) it is to use Python to interactively explore and analyze Twitter data. We started out the chapter by learning how to create an authenticated connection and then progressed through a series of examples that illustrated how to discover trending topics for particular locales, how to search for tweets that might be interesting, and how to analyze those tweets using some elemtnary but effective techniques. Even what seemed like a somewhat arbitrary trending topic turned out to lead us down interesting paths with lots of possibilities for additional analysis.

One of the primary takeaways from this chapter from an analytical standpoint is that counting is generally the first step to any kind of meaningful quantitative anlaysis. Although frequency analysis is very simple, yet very powerful, tool for your repertoire that shouldn't be overlooked just because it's so obvious. On the contrary, frequency analysis and measures such as lexical diversity should be employed early and often for precisely the reason that it's so obvious and simple. Often times, but not always, the simplest techniques can rival the quality of analytics for more sophisticated analysis. With respect to data in the Twitterverse, these modest techniques can usually get you quite a long way toward answering the question, "What are people talking about right now?" Now that's something we'd all like to know, isn't it?

[XXX: xref Part II] of this book contains a number of Twitter recipes that you may find are interesting, useful, and will help you answer that question if you enjoy mining Twitter data.

The source code foutlined ior this chapter and all other chapters is available online [XXX: add link to repo] as ordinary Python code as well as in a convenient iPython Notebook format that you're highly encouraged to try out from the comfort of your own web browser.

Recommended Exercises

- Bookmark and spend some time reviewing Twitter's API documentation. In particular, spend some time browsing the information on the REST API and Platform Objects.
- If you haven't already, get comfortable working in iPython as a more productive alternative to the traditional Python interpreter. Over the course of your social web mining career, the saved time and increased productivity will really start to add up.
- If you have a Twitter account with a non-trivial number of tweets, request your historical tweet archive from your account settings and analyze it. The export of your account data includes files organized by time period that are in a convenient JSON format. See the README.txt file included in the downloaded archive for more details. What are the most common terms that appear in your tweets? Who do you retweet the most often? How many of your tweets are retweeted (and why do you think this is the case?)
- Take some time to explore Twitter's REST API with its developer console. Although we opted to dive in with the `twitter` Python package in a programmatic fashion in this chapter, the console can still be useful for exploring the API, the effects of parameters, etc. The command line tool Twurl is another option to consider if you prefer working in a terminal.
- Complete the exercise of determining whether or not there seems to be a spiritual or religious affiliation for the users who retweeted the status citing "God" as someone important to them, or follow the workflow in this chapter but for a trending topic or arbitrary search query of your own choosing. Explore some of the advanced search features that are available for more precise querying.
- Explore Yahoo! GeoPlanet's Where On Earth ID API so that you can compare and contract trends from different locales.
- Take an closer look at `matplotlib` and learn how to create beautiful plots of 2-D and 3-D data with IPython Notebook.

- Explore and apply some of the exercises from [XXX: xref Part II] of this book.

CHAPTER 2

Mining Facebook's Social Graph: Analyzing Fan Pages, Examining Friendships, and More

In this chapter, we'll tap into the Facebook Platform through the (Social) Graph API and explore some of the vast possibilities. Facebook is arguably the heart of the social web and is somewhat of an all-in-one wonder given that more than half of its 1 billion users[1] are active each day updating statuses, posting photos, exchanging messages back and forth, chatting in realtime, checking in to physical locales, playing games, shopping, and just about anything else you can imagine. From a social web mining standpoint, the wealth of data that Facebook stores about individuals, groups, and products is quite exciting, because Facebook's clean API presents incredibly valuable opportunities to synthesize it into the world's most precious commodity--information-- and glean valuable insights. On the other hand, this great power commands great responsibility, and Facebook has instrumented the most sophisticated set of online privacy controls that the world have ever seen in order to help protect its users from exploit. It's worth noting that although Facebook is self-proclaimed as a social graph, it's been steadily transcending into a very valuable interest graph as well, because it maintains relationships between people and things that they're interested in through its fan pages and "likes" feature. In this regard, you may sometimes hear it referred to as a "social interest graph."

1. Internet usage statistics show that the world population is estimated to be approximately 7 billion with an estimated number of internet users being almost 2.5 billion.

For the most part, you can make a case that interest graphs implicitly exist and can be bootstrapped from most sources of social data. As an example, you could make the case that Twitter is actually an interest graph because of its asymmetric "following" (or, to say it another way, "interested in") relationships between people and other people, places, or things, or you might use membership in a LinkedIn group to associate a person with an interest. We'll return to the idea of bootstrapping an interest graph from social data in [XXX: xref GitHub chapter].

The remainder of this chapter assumes that you have an active Facebook account, which is required to gain access to the Facebook APIs. Although there are plenty of fun things that you can do to analyze public information, you may find that it's a little bit more fun if you are analyzing data from your own social network, so it's worth adding a few friends if you are new to Facebook.

The Facebook Platform

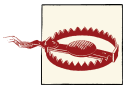
The Social Graph is the heart of the Facebook Platform, but before engaging in a more narrowly focused discussion about the Social Graph itself, it's worthwhile to briefly discuss a broader overview of the Facebook Platform in general. The Facebook Platform is a mature, robust, and well-documented gateway into a walled garden of what may be the most comprehensive and well organized information store ever amassed both in terms of breadth and depth. It's is broad in that its user base represents about one-seventh of the entire living population, and it's deep with respect to the amount of information that's known about any one of its particular users. Whereas Twitter features an asymmetric friendship model that is open and predicated on following other users without any particular consent, Facebook's friendship model is symmetric and requires a mutual agreement between users to gain visibility into one another's interactions and activities. Furthermore, whereas virtually all interactions except for private messages between users on Twitter are public statuses, Facebook allows for much more finely-grained privacy controls in which friendships can be organized and maintained as lists with varying levels of visibility available to a friend on any particular activity. For example, you might choose to only share a link or photo with a particular list of friends as opposed to your entire social network.

As a social web miner, the only way that you can access a Facebook user's account data is by registering an application and using that application as the entry point into the Facebook developer platform. Moreover, the only data that's available to an application is whatever the user has explicitly authorized it to access. For example, as a developer writing a Facebook application, you'll be the user that's logging into the application, and the application will be able to access any data that you explicitly authorize it to access. In that regard, a Facebook user might think of an application a bit like any of its Facebook

friends in that the user is ultimately in control of what the application can access and is able to revoke access at any time. The Facebook Platform Policies document is a must-read for any Facebook developer as it provides the comprehensive set of rights and responsibilities for all Facebook users as well as the spirit and letter of the law for Facebook developers. If you' haven't already, it's worth taking a moment to review Facebook's developer policies as well as bookmark the Facebook Developers homepage since it is the definitive entry point into the Facebook Platform and its documentation.

Keep in mind that as a developer mining your own account, you may not have a problem allowing your own application to access all of your account data. Beware, however, of aspiring to develop a successful hosted application that requests access to beyond the minimum amount of data necessary to complete, because it's quite likely that a user will not recognize or trust your application to command that level of privilege (and rightly so.)

Although we'll programatically access the Facebook Platform later in this chapter, Facebook provides a number of useful developer tools including a Graph API Explorer app that we'll be using for initial familiarization with the Social Graph. It provides an intuitive and turn-key way of querying the Social Graph, and once you're comfortable with how the Social Graph works, translating queries into Python code for automation and further processing comes quite naturally. Although we'll work though the Graph API as part of the discussion, you may benefit from an initial review of the well written Getting Started: The Graph API document as a comprehensive preamble.



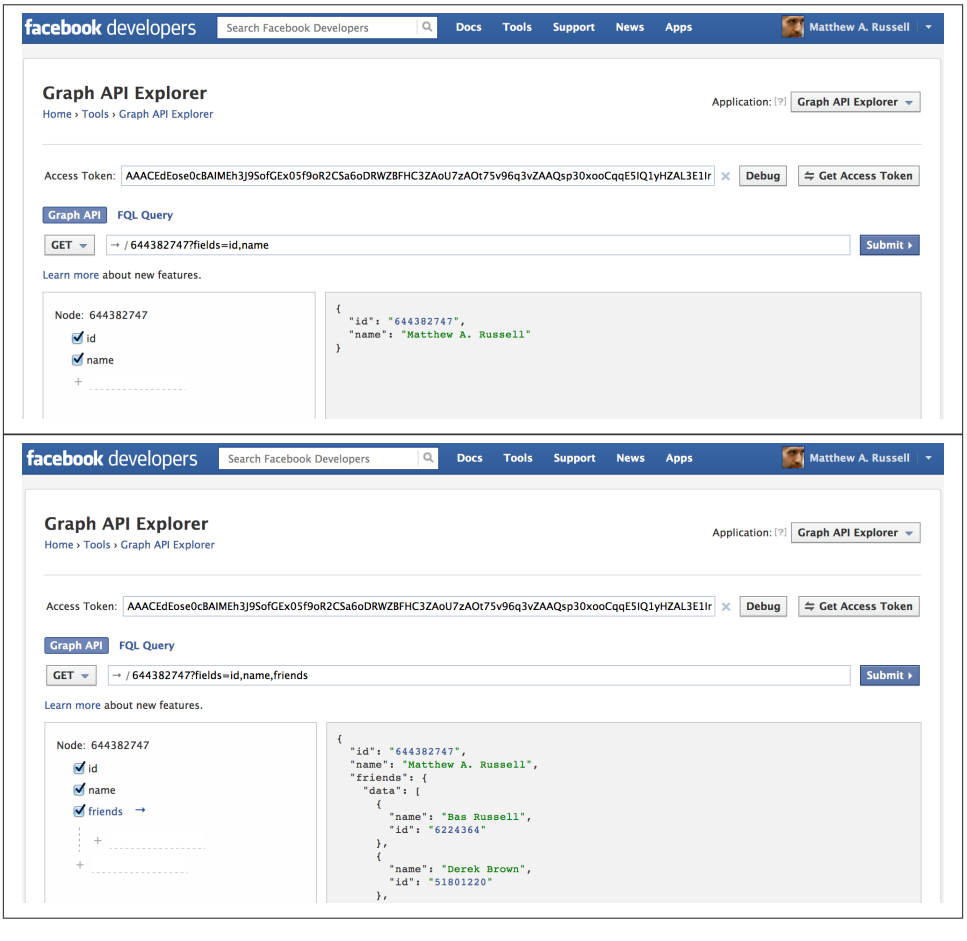
In addition to the Graph API, you may also encounter the Facebook Query Language (FQL) and what is now referred to as the Legacy REST API. Be advised that although the FQL is still very much alive, the Legacy REST API is in deprecation and will be phased out soon. Do not use it for any development that you do with Facebook.

Exploring the Graph API

As its name implies, the Social Graph is a massive graph data structure representing social interactions and consists of nodes and connections between the nodes. The Graph API provides the primary means of interacting with the Social Graph, and the best way to get acquainted with the Graph API is to spend a few minutes tinkering around with the Graph API Explorer. Figure 2-1 illustrates a progressive series of Graph API queries that resulted from clicking on the "+" symbol and adding additional connections and fields. There are a few notable items about this particular query:

- The access token that appears in the application is an OAuth token that is provided as a courtesy for the logged in user and is the same OAuth token that your application would need to access the data in question. We'll come back to the process for registering an application and getting your own OAuth token in the next section. If this is your first encounter with OAuth, it's probably sufficient at this point to know that the protocol is a social web standard and stands for Open Authorization. In short, OAuth is a means of allowing users to authorize 3rd party applications to access their account data without needing to share sensitive information like a password. See (to come) for more information about OAuth.
- The basis of the query is a node with an id (identifier) of "644382747", corresponding to a person named "Matthew A. Russell", who is preloaded as the the currently logged in user for the Graph Explorer, and the "id" and "name" values for the node are called fields. The basis of the query could just as easily have been any other node, and as we'll soon see, it's very natural to "walk" or traverse the graph and query other nodes that may be people or other kinds of things such as books, TV shows, etc.
- Modifying the original query with a "friends" connection is achieved by clicking on the "+" and adding "friends" from the popup menu. The "friends" connections that appear in the console represent nodes that are connected to the original query node. At this point, you could click on any of the blue id fields in these nodes and initiate a query with that particular node as the basis. In network science terminology, we now have what is called an "ego graph", because it has an actor or "ego" as its focal point or logical center that is connected to other nodes around it. An ego graph resembles a hub and spokes if you were to draw it neatly on a piece of paper.
- A further modification to the original query is to add additional "likes" connections for each of your friends. Before you can retrieve "likes" connections for your friends, however, you must authorize the Graph API Explorer application to explicitly access your friends' "Likes" by updating the access token that it uses and then approving this access as shown in Figure 2-2. The Graph API Explorer allows you to easily authorize it by clicking on the "Get Access token" button and checking the "friends_likes" box on the "Friends Data Permissions" tab. In network science terminology, we still have an ego graph, but it's potentially much more complex at this point, because of the many additional nodes and connections that could exist amongst them.
- The "Debug" button can be useful for troubleshooing queries that you think should be returning data but aren't actually returning it based on the authorizations associated with the access token.
- The results of a Graph API query are returned in a convenient JSON format that can be easily manipulated and processed.

It is of particular note that the Graph API Explorer is not a particularly special tool of any particular kind; aside from being able to pre-populate and debug your access token, it is an ordinary Facebook app that uses the same developer APIs that any other developer application would use. In fact, the Graph API Explorer is handy when you have a particular OAuth token that's associated with a particular set of authorizations for application that you are developing and you want to run some queries as part of a exploratory development effort or debug cycle. We'll revisit this general idea shortly as we programatically access the Graph API.



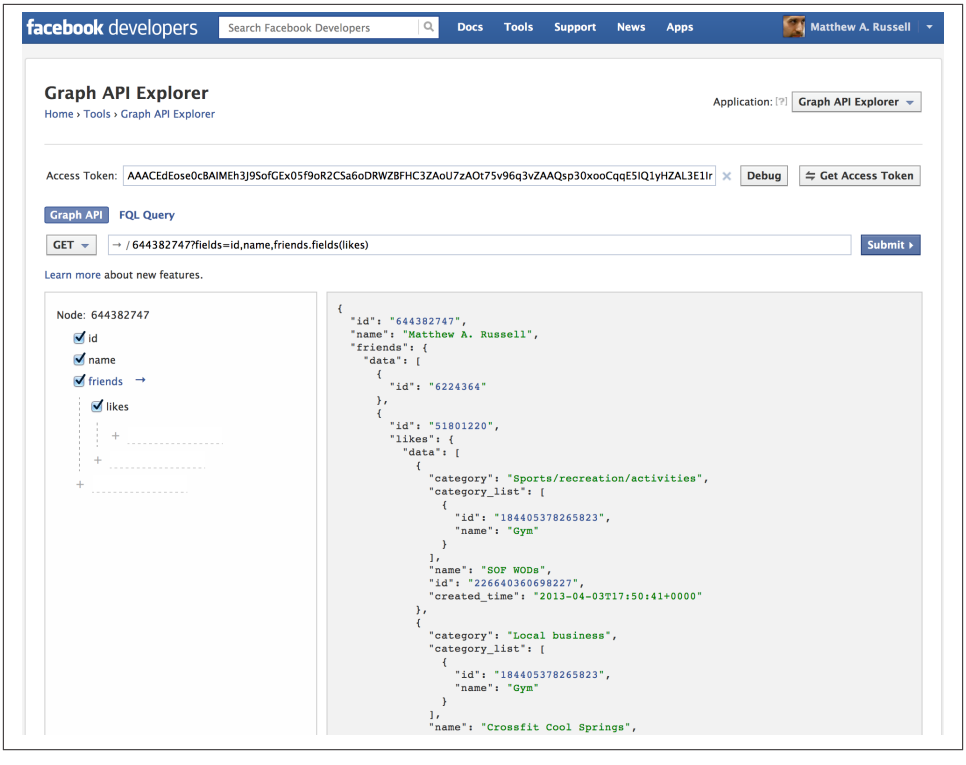


Figure 2-1. Using the Graph API Explorer application to progressively build up a query for friends' interests. Top: A query for a node. Middle: A query for a node and connections to friends. Bottom: A query for a node, connections to friends, and likes for those friends.

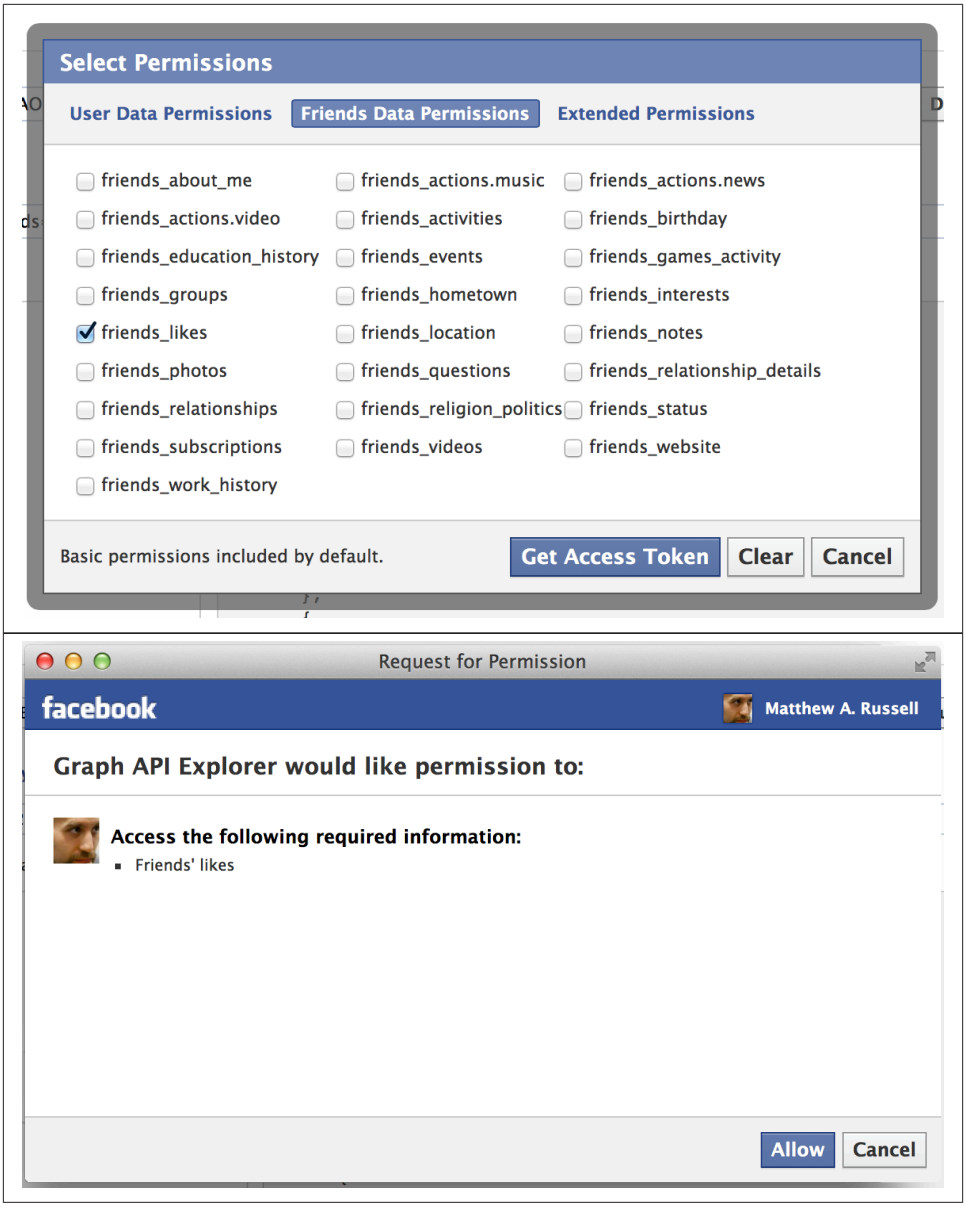


Figure 2-2. Facebook applications must explicitly request authorization to access a user's account data. Top: The Graph API Explorer permissions panel. Bottom: A Face-

book dialog requesting authorization for the Graph API Explorer application to access friends' "likes" data.

Facebook Query Language (FQL)

In addition to the Graph API, FQL provides a fine alternative way to query Facebook's Social Graph and has an SQL inspired syntax that most developers find intuitive. It seems to be the case that any data you could query with the Graph API, you could also query via FQL, and although it may be the case that some advanced queries that are possible with FQL may not be possible with the Graph API, it appears that Facebook's longer term plan is to ensure that the Graph API is at full parity with FQL. For example, some recent investments in the Graph API resulted in a number of powerful new features such as field expansion and nesting. In case you're interested in learning more about FQL, consult the FQL Reference, and try out a query with the FQL Query console that's available as an alternate option from the Graph API Explorer. For example, you could query the first and last names of your friends in the "FQL Query" tab of the Graph API Explorer with the following FQL query:

```
select first_name, last_name
from user
where uid in (
  select uid2
  from friend
  where uid1 = me()
)
```

Although we'll programatically explore the Graph API with a Python package later in this chapter, you could opt to make Graph API queries more directly over HTTP yourself by mimicing the request that you see in the Graph API Explorer should you desire to do so. For example, Example 2-1 uses the `requests` package to simplify the process of making an HTTP request (as opposed to using a much more cumbersome package from Python's standard library such as `urllib2`) for fetching your friends and their likes. The query is driven by the values in the `fields` parameter and is the same as what would be built up interactively in the Graph API Explorer. Of particular interest is that the `friends.fields(id,name,likes)` syntax uses a relatively new feature of the Graph API called field expansion that is designed to make multiple queries in a single API call.

Example 2-1. Making Graph API requests over HTTP

```
import requests # pip install requests

# XXX: Use your access token from the Graph API Explorer
ACCESS_TOKEN = ''

base_url = 'https://graph.facebook.com/me'
fields = 'id,name,friends.fields(id,name,likes)'
```

```
r = requests.get('%s?fields=%s&access_token=%s' % \
    (base_url, fields, ACCESS_TOKEN,))

# Deseriealize the response and write back out as
# pretty-printed JSON
print json.dumps(json.loads(r.content), indent=1)
```

It appears as though the default limits for queries at the time of this writing is to return up to 5,000 items. It's possible but somewhat unlikely that you'll be making Graph API queries that could return more than 5,000 items, but if you do, consult the pagination documentation for information on how to navigate through the "pages" of results.

Open Graph Protocol

In addition to sporting a powerful Graph API that allows you to traverse the Social Graph we've been discussing that allows for querying familiar Facebook objects, you should also know that Facebook unveiled something called the Open Graph protocol (OGP) back in April 2010 at the same F8 conference that it introduced the Social Graph. In short, OGP is a mechanism that enables developers to make any web page an object in Facebook's Social Graph by injecting some RDFa metadata into the page. Thus, in addition to being able to access the dozens of objects familiar to the Facebook user experience from within its "walled garden" that are described in the Graph API Reference (users, pictures, videos, checkins, links, status messages, etc.) you might also encounter pages from the Web that represent meaningful concepts that have been grafted into the Social Graph. In other words, OGP is a means of "opening up" the Social Graph, and you'll see these concepts described in Facebook's developer documentation as its "Open Graph"[2].

There are practically limitless options for leveraging OPG to graft web pages into its Social Graph in interesting and valuable ways, and the chances are very good that you've already encountered many of them and not even realized it. For example, consider Figure 2-3, which illustrates a page for the movie The Rock from IMDb.com. In the sidebar to the right, you see the a rather familiar looking Like button with the message, "19,319 people like this. Be the first of your friends." message. As it turns out, the way that IMDb enables this functionality is by implementing OGP for each of its URLs that correspond to objects that would be relevant for inclusion in the Social Graph. With the right RDFa metadata in the page, Facebook is then able to unambiguously enable "con-

2. Throughout this section describing how the implementation of OGP, the term Social Graph is generically used as opposed to continually distinguishing between Social Graph and Open Graph unless explicity emphasized otherwise.

nections" to these objects and incorporate them into activity streams and other key elements of the Facebook user experience.

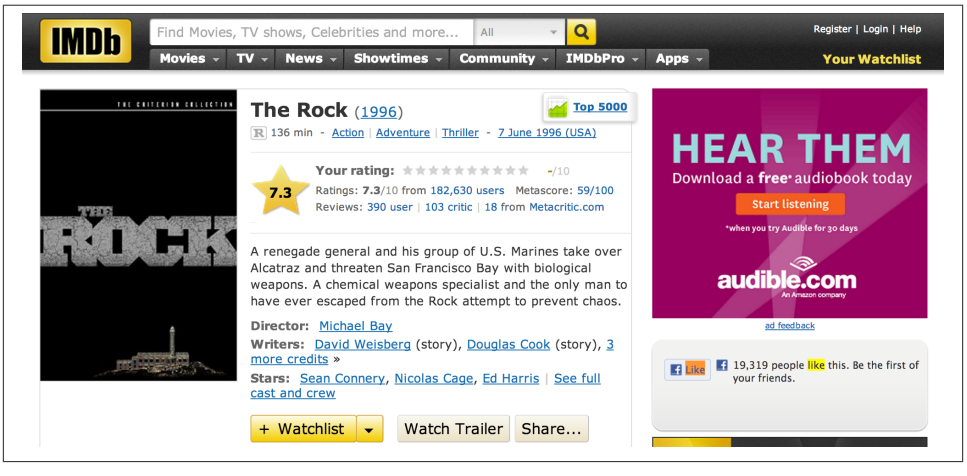


Figure 2-3. An IMDb page featuring an implementation of OGP for The Rock

Implementation of OGP manifesting as the Like button on web pages may seem a bit obvious or if you're used to seeing them for the past few years, but the fact that Facebook has been fairly successful at opening up its own development platform in a way that allows for arbitrary inclusion of objects on the Web is rather profound and has some potentially very significant consequences. For example, at of the time of this writing in early 2013, Facebook has just started the process of launching its new Graph Search product to a limited audience. Whereas companies like Google crawl and index the entire Web in order to enable search, the basic idea behind Facebook's Graph Search is that you would type something into a search box just like the typical Google user experience, but get back results that are inherently personalized to you based upon the vast amount of information that Facebook knows about you. The interesting rub now that OGP is fairly well established is that Facebook's Graph Search results wouldn't be limited to things within the Facebook user experience since connections from the Web are inherently incorporated into the Social Graph. It's out of scope to ponder the wider ramifications of how disruptive Graph Search could be to the Web given Facebook's user base but a though exercise well worth the time to consider.

Let's briefly take a look at the gist of implementing OGP before moving on to Graph API queries. Example 2-2 is the canonical example from the OGP documentation that demonstrates how to turn the IMDB's page on The Rock into an object in the Open Graph protocol as part of an XHTML document that uses namespaces. These bits of metadata have great potential once realized at a massive scale, because they enable a URI like *http://www.imdb.com/title/tt0117500* to unambiguously represent any web

page—whether it's for a person, company, product, etc.—in a machine-readable way and furthers the vision for a semantic web. for In addition to being able to "like" The Rock, users could potentially interact with this object in other ways through custom actions. For example, users might be able to indicate that they have watched The Rock since it is a movie. OGP allows for a wide and flexible set of actions between users and objects as part of the Social Graph.

If you haven't already, go ahead and view the source HTML for *http://www.imdb.com/title/tt0117500* and see what the RDFa looks like out in the wild for yourself.

Example 2-2. Sample RDFa for the Open Graph protocol

```
<html xmlns:og="http://ogp.me/ns#">
<head>
<title>The Rock (1996)</title>
<meta property="og:title" content="The Rock" />
<meta property="og:type" content="movie" />
<meta property="og:url" content="http://www.imdb.com/title/tt0117500/" />
<meta property="og:image" content="http://ia.media-imdb.com/images/rock.jpg" />
...
</head>
...
</html>
```

At its core, querying the Graph API for Open Graph objects is incredibly simple: substitute an object's ID in the URI *http(s)://graph.facebook.com/ID* to fetch details about the object. For example, fetching the URL *http://graph.facebook.com/http://www.imdb.com/title/tt0117500* in your web browser would return the response in Example 2-3.

Example 2-3. A sample response for an Open Graph query

```
{
   "id": "114324145263104",
   "name": "The Rock (1996)",
   "picture": "http://profile.ak.fbcdn.net/hprofile-ak-snc4/hs344.snc4/41581...jpg",
   "link": "http://www.imdb.com/title/tt0117500/",
   "category": "Movie",
   "description": "Directed by Michael Bay.  With Sean Connery, Nicolas Cage, ...",
   "likes" : 3
}
```

If you inspect the source for the URL *http://www.imdb.com/title/tt0117500*, you'll find that fields in the response correspond to the data in the `meta` tags of the page, and this is no coincidence. The delivery of rich metadata in response to a simple query is the whole idea behind the way OGP is designed to work. Where it gets more interesting is

when you explicitly request additional metadata for an object in the page by appending the query string parameter `metadata=1` to the request. A sample response for the query *https://graph.facebook.com/114324145263104?metadata=1* is shown in Example 2-4.

Example 2-4. A sample response for an Open Graph query with the optional metadata included


```
{
   "id": "118133258218514",
   "name": "The Rock (1996)",
   "picture": "http://profile.ak.fbcdn.net/hprofile-ak-snc4/..._s.jpg",
   "link": "http://www.imdb.com/title/tt0117500",
   "category": "Movie",
   "website": "http://www.imdb.com/title/tt0117500",
   "description": "Directed by Michael Bay.  With Sean Connery, Nicolas Cage, ...",
   "likes": 3,
   "metadata": {
      "connections": {
         "feed": "http://graph.facebook.com/http://www.imdb.com/title/tt0117500/feed",
         "posts": "http://graph.facebook.com/http://www.imdb.com/title/tt0117500/posts",
         "tagged": "http://graph.facebook.com/http://www.imdb.com/title/tt0117500/tagged",
         "statuses": "http://graph.facebook.com/http://www.imdb.com/title/...",
         "links": "http://graph.facebook.com/http://www.imdb.com/title/tt0117500/links",
         "notes": "http://graph.facebook.com/http://www.imdb.com/title/tt0117500/notes",
         "photos": "http://graph.facebook.com/http://www.imdb.com/title/tt0117500/photos",
         "albums": "http://graph.facebook.com/http://www.imdb.com/title/tt0117500/albums",
         "events": "http://graph.facebook.com/http://www.imdb.com/title/tt0117500/events",
         "videos": "http://graph.facebook.com/http://www.imdb.com/title/tt0117500/videos"
      },
      "fields": [
         {
            "name": "id",
            "description": "The Page's ID. Publicly available. A JSON string."
         },
         {
            "name": "name",
            "description": "The Page's name. Publicly available. A JSON string."
         },
         {
            "name": "category",
            "description": "The Page's category. Publicly available. A JSON string."
         },
         {
            "name": "likes",
            "description": "\\* The number of users who like the Page..."
         }
      ]
   },
   "type": "page"
}
```

The items in `metadata.connections` are pointers to other nodes in the graph that you can crawl to get to other interesting bits of data. For example, you could follow the "photos" link to pull down photos associated with the movie, and potentially walk links associated with the photos to discover who posted them or see comments that might have been made about them. In case it hasn't already occurred to you, you are also an object in the graph. Try visiting the same URL prefix, but substitute in your own Facebook ID or username as the URL context and see for yourself.

As a final note of advice before moving on to programmatically accessing the Graph API, when considering the possibilities with OGP, be forward-thinking and creative, but bear in mind that it's still new in the grand scheme of things and continues to evolve. As it relates to the semantic web and web standards in general, consternation about the use of "open" has understandably surfaced, various kinks in the spec have been worked out along the way, and some are still probably being worked out. You could also make the case that OGP is essentially a single-vendor effort, and it's little more than on par with the capabilities of meta elements from the much earlier days of the Web, although the social effects appear to be driving a very different outcome. In effect, OGP and the possibilities for Graph Search may really more of snowflake than a standard at this moment, but the potential is certainly in place for that to change, the indicators for its success are trending in a positive direction, and many exciting things may happen as the future unfolds and innovation continues to take place. Let's now turn back and hone in on how to access the Graph API to work now that you have an appreciation for the fuller context of the Social Graph.

Accessing the Graph API

This section transitions from our exploration of the Graph API with the Graph API Explorer application by providing a brief overview of how to create a Facebook application, retrieve an access token with OAuth, and access the Graph API programatically from Python. We'll keep the discussion somewhat focused on retrieving and analyzing Facebook data and won't get into the nuances of building a Facebook application: building Facebook applications is a topic worthy of a book or two in and of itself, and besides, there are plenty of tutorials online that you can easily find to help you build an app once you are ready.

As with any other application that depends on OAuth, you'll need to opt into the developer community, create an application, and acquire some credentials in order to get started. Although you could opt to get an access token without implementing an OAuth workflow through Facebook's Access Token Tool that's designed to ease the test cycle for developers, it's not very difficult and very worthwhile to implement a simple script for retrieving an access token since OAuth is an integral part of the social web. The following list summarizes the main steps involved:

- Go to *https://developers.facebook.com/apps* and create an application by clicking on the "+ Create New App" button and filling in the application name and namespace as shown in

- Take note of your newly created application's "App ID/API Key" value, which is the credentials that are used as part of the OAuth flow for fetching an access token.
- In your application's "Basic" settings, you'll need to specify a URL that Facebook redirects to once a user authorizes your application to access account data. For now, set a value of http://localhost:5000/oauth_redirect for the "Site URL" under the "Website with Facebook Login" pane. Facebook redirects here, and we'll be running a local web server to capture the request and parse the access token out of the fragment identifier (the part of the URL that appears after a hash symbol) in the URL. Even without a web server running to intercept the redirect, however, it would still be possible to carefully copy and paste the access token if needed.

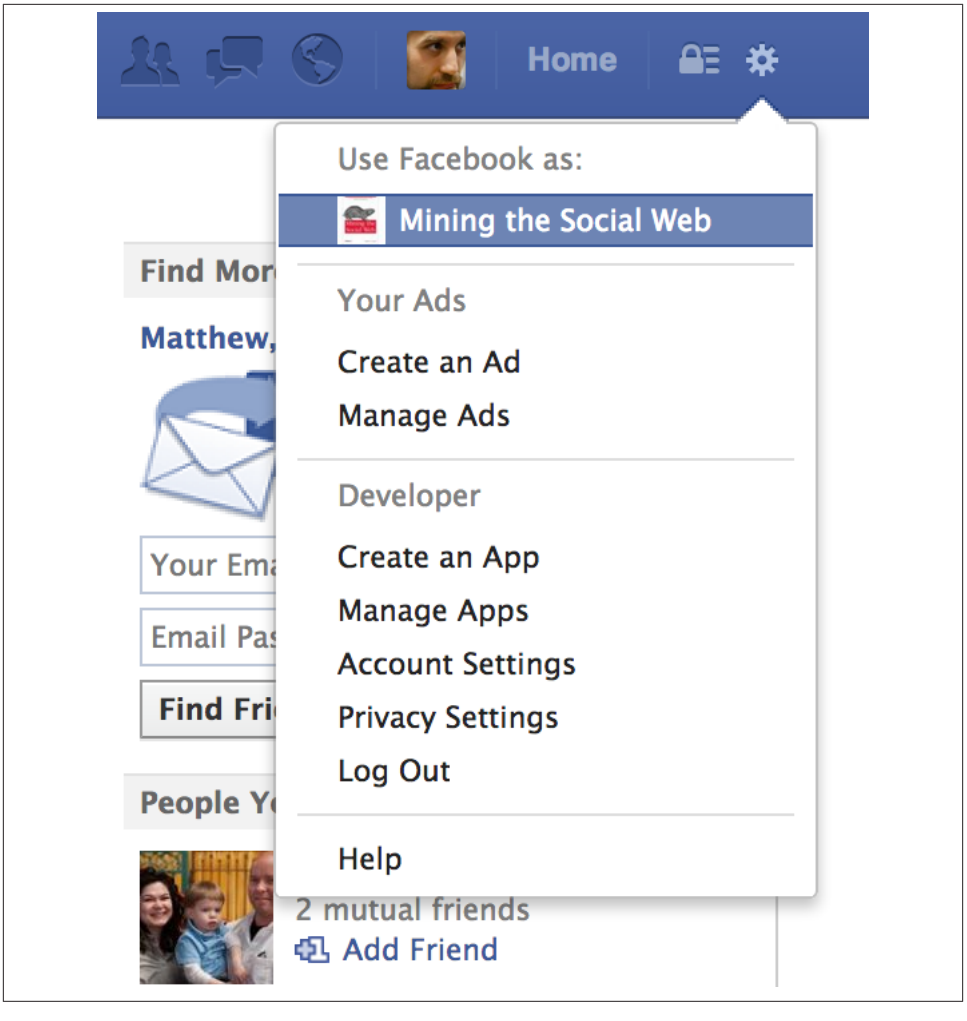


Figure 2-4. Create an application on Facebook by selecting "Create an App" from the "Developer" section of your account settings from the main menu bar, or by navigating directly to https://developers.facebook.com/apps

With the basic details of application registration out of the way, the next step is executing the script shown in Example 2-5 that handles authorization and gets you the access token to access the API as described in the developer docs for the OAuth Dialog. Although there's a bit of code involved, the essence of the script is very simple: open a web browser and direct you to sign into your Facebook account so that you can authorize the application, and then parse out the access token from Facebook's redirect back a local web server called Flask. All that you'll need to do is `pip install flask` in a terminal to get Flask ready to go. The `webbrowser` package that is also introduced as

part of this workflow is part of Python standard library and can be used to automatically launch the browser as an added convenience.

The process of starting up the local web server itself is really just a few lines of code, but a couple of extra steps have been taken to ensure that no race conditions occur between the act of opening a web browser and the web server starting up and handling the redirect. Although not absolutely necessary for this script to work, the web server is also configured to shut itself down once it serves a request so that the script can finish execution cleanly.

Example 2-5. Getting an OAuth token for a Facebook application

```
import webbrowser
import urllib
from flask import Flask, request
from threading import Timer

# XXX: Get this value from your Facebook application's settings for the OAuth flow

CLIENT_ID = ''

# This value is where Facebook will redirect. We'll configure an embedded
# web server to be serving requests here

REDIRECT_URI = 'http://localhost:5000/oauth_helper'

# You could customize which extended permissions are being requested for your app
# by adding additional items to the list below. See
# https://developers.facebook.com/docs/reference/login/extended-permissions/

EXTENDED_PERMS = ['user_likes', 'friends_likes']

# Configure an emedded web server that accepts one request, displays a message
# by parsing the fragment identifier out of the browser window and shuts
# itself down

webserver = Flask("OAuthHelper")
@webserver.route("/oauth_helper")
def oauth_helper():
    shutdown_after_request = request.environ.get('werkzeug.server.shutdown')
    shutdown_after_request()
    return '''<script type="text/javascript">
    var at = window.location.hash.substring("access_token=".length+1).split("&")[0];
    document.write("Access Token: <strong>" + at + "</strong>");
    </script>'''

# Send an OAuth request to Facebook, handle the redirect, and display the access
# token that's included in the redirect for the user to copy and paste

def login():
```

```
    args = dict(client_id=CLIENT_ID, redirect_uri=REDIRECT_URI,
                scope=','.join(EXTENDED_PERMS), type='user_agent', display='popup'
                )

    # Open a browser after a 1 second delay to prevent a possible race condition
    # and ensure that the web server is running and can handle the request

    oauth_url = 'https://facebook.com/dialog/oauth?' + urllib.urlencode(args)
    Timer(1, webbrowser.open, args=[oauth_url]).start()

    # Run the server to accept the redirect back from Facebook and capture the access
    # token. This command blocks, but the web server is configured to shut itself down
    # after it serves a request, so after the redirect occurs, program flow will continue

    webserver.run()

login()
```

One important detail you're probably wondering about is about the possibilities for the values in `EXTENDED_PERMS`, and a brief explanation is certainly in order. The first time you try to log in to the application, it'll notify you that the application is requesting your likes and your friends' likes, but the possibilities of what you could actually request are vast. The details of all of the extended permissions that are available are described in Facebook's authentication documentation, but the short story is that, by default, applications can only access some basic data from user profiles such as name, gender, and friends. Explicit permissions must be granted to access additional data beyond these minimums. It's been mentioned already but an important subtlety to observe here is that you might be able to see certain details about your friends, such as things that they "like" or activity on their walls through your normal Facebook account, but your app cannot access these same details unless you have granted it explicit permission to do so. In other words, there's a difference between what you'll see in your friends' profiles when signed in to facebook.com and the data that your app will get back when requesting information through the API. This is because it's Facebook (not you) who is exercising the platform to get data when you're logged in to facebook.com, and that's different your app that is requesting data. If your app does not request extended permissions to access data but tries to access it anyway, you may get back empty data objects as opposed to an explicit error message of any kind. The Access Token tool can be helpful for debugging these scenarios.

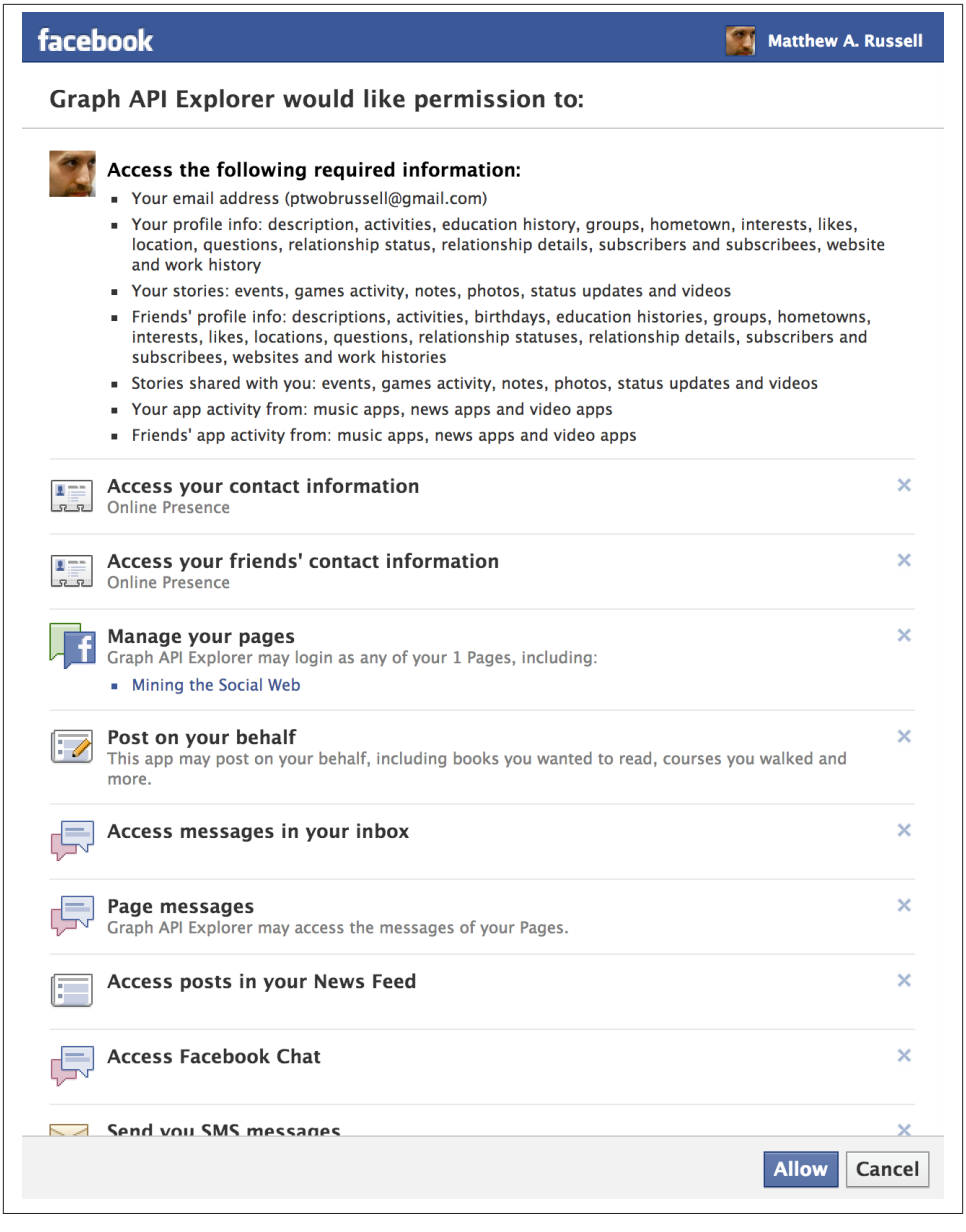
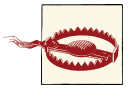


Figure 2-5. Don't do this: the rather scary dialog a user would see if an application requested permission to access to virtually everything available to it, and this isn't even all of it. (There's a scroll bar that you don't see.)



The Facebook platform's documentation can still be a bit ambiguous in a few places, and it may not tell you everything you need to know about extended permissions. For example, it appears that in order to access religious or political information for friends (your friends_religion_politics extended permission), those friends must have set it to be publicly viewable or have your app installed and have explicitly authorized access to this data as well via their user_religion_politics extended permission.

Now, outfitted with a shiny new access token and the know-how to access all of the data available to you from your own account, it's time to move on to Graph API queries.

Exploring the Graph API one connection at a time

An official Python SDK for the Graph API is a community-fork of that repository previously maintained by Facebook and can be installed per the standard protocol with `pip` as follows: `pip install facebook-sdk`. This package contains a few useful convenience methods that allow you to interact with Facebook in a number of ways including the ability to make FQL queries, post statuses or photos, etc. However, there are really just a few key methods from the `GraphAPI` class (defined in the *facebook.py* source file) that you need to know about in order to use the Graph API to fetch data as shown below, so you could just as easily opt to query over HTTP directly with `requests` as was illustrated in Example 2-1 if you prefer.

- `get_object(self, id, **args)`

  Example usage: `get_object("me", metadata=1)`
- `get_objects(self, id, **args)`

  Example usage: `get_objects(["me", "some_other_id"], metadata=1)`
- `get_connections(self, id, connection_name, **args)`

  Example usage: `get_connections("me", "friends", metadata=1)`
- `request(self, path, args=None, post_args=None)`

  Example usage: `request("search", {"q" : "social web", "type" : "page"})`

Unlike with other social networks, there don't appear to be clearly published guidelines about Facebook API rate limits. Although the availability of the APIs seems to be quite generous, you should still carefully design your application to use the APIs as little as possible and handle any/all error conditions as a recommended best practice.

The most common (and often, the only) keyword argument you'll probably use is `metadata=1`, in order to get back the connections associated with an object in addition to just the object details themselves. Take a look at Example 2-6, which introduces the `GraphAPI` class and uses its `get_objects` method to query for information about you, your friends, and "social web". This example also introduces a helper function called `pp` that is used throughout the remainder of this chapter for pretty-printing results as nicely formatted JSON.

Example 2-6. Querying the Graph API with Python

```
import facebook
import json

# A helper function to pretty print Python objects as JSON

def pp(o):
    print json.dumps(o, indent=1)

# Create a connection to the Graph API with your access token

g = facebook.GraphAPI(ACCESS_TOKEN)

# Execute a few sample queries

print '---------------'
print 'Me'
print '---------------'
pp(g.get_object('me'))
print
print '---------------'
print 'My Friends'
print '---------------'
pp(g.get_connections('me', 'friends'))
print
print '---------------'
print 'Social Web'
print '---------------'
pp(g.request("search", {'q' : 'social web', 'type' : 'page'}))
```

Sample results for the queries from Example 2-6 are presented in Example 2-7, and the results are hopefully predictable enough. You get back exactly what you'd expect to see,

and if you were using the Graph API Explorer, the results would be identical. During development, it can often be very handy to use the Graph API Explorer and an IPython or IPython Notebook in tandem with one another depending on exactly what objective you're trying to accomplish. The advantage of the Graph API Explorer is the ease with which you can click on id values and spawn new queries during exploratory efforts.

Example 2-7. Sample results from Example 2-6

```
---------------
Me
---------------
{
 "last_name": "Russell",
 "relationship_status": "Married",
 "locale": "en_US",
 "hometown": {
  "id": "104012476300889",
  "name": "Princeton, West Virginia"
 },
 "quotes": "The only easy day was yesterday.",
 "favorite_athletes": [
  {
   "id": "112063562167357",
   "name": "Rich Froning Jr. Fan Site"
  }
 ],
 "timezone": -5,
 "education": [
  {
   "school": {
    "id": "112409175441352",
    "name": "United States Air Force Academy"
   },
   "type": "College",
   "year": {
    "id": "194603703904595",
    "name": "2003"
   }
  }
 ],
 "id": "644382747",
 "first_name": "Matthew",
 "middle_name": "A.",
 "languages": [
  {
   "id": "106059522759137",
   "name": "English"
  },
  {
   "id": "312525296370",
   "name": "Spanish"
  }
```

```
 ],
 "location": {
  "id": "103078413065161",
  "name": "Franklin, Tennessee"
 },
 "email": "ptwobrussell@gmail.com",
 "username": "ptwobrussell",
 "bio": "How I Really Feel About Using Facebook (Or: A Disclaimer)...",
 "birthday": "06/17/1981",
 "link": "http://www.facebook.com/ptwobrussell",
 "verified": true,
 "name": "Matthew A. Russell",
 "gender": "male",
 "work": [
  {
   "position": {
    "id": "135722016448189",
    "name": "Chief Technology Officer (CTO)"
   },
   "start_date": "0000-00",
   "employer": {
    "id": "372007624109",
    "name": "Digital Reasoning"
   }
  }
 ],
 "updated_time": "2013-04-04T14:09:22+0000",
 "significant_other": {
  "name": "Bas Russell",
  "id": "6224364"
 }
}

---------------
My Friends
---------------
{
 "paging": {
  "next": "https://graph.facebook.com/644382747/friends?...",
 },
 "data": [
  {
   "name": "Bas Russell",
   "id": "6224364"
  },
  ...
  {
   "name": "Jamie Lesnett",
   "id": "100002388496252"
  }
 ]
}
```

```
---------------
Social Web
---------------
{
 "paging": {
  "next": "https://graph.facebook.com/search?q=social+web&type=page...",
 },
 "data": [
  {
   "category": "Book",
   "name": "Mining the Social Web",
   "id": "146803958708175"
  },
  {
   "category": "Internet/software",
   "name": "Social & Web Marketing",
   "id": "172427156148334"
  },
  {
   "category": "Internet/software",
   "name": "Social Web Alliance",
   "id": "160477007390933"
  },
  ...
  {
   "category": "Local business",
   "name": "Social Web",
   "category_list": [
    {
     "id": "2500",
     "name": "Local Business"
    }
   ],
   "id": "145218172174013"
  }
 ]
}
```

At this point, you have the power of the Graph API Explorer and the power of a Python console and all that it has to offer at your fingertips. Now that we've scaled the walled garden, let's now turn our attention to analyzing some of its data.

Analyzing Facebook Pages

Given that this very book has a corresponding Facebook page that happened to turn up as the top result in the search for "social web", it seems natural enough that we could use

it as an illustrative starting point for some instructive analysis here in this chapter[3]. Here are just a few questions that might be interesting to consider of this book's Facebook page or just about any other Facebook page:

- How popular is the page?
- How engaged are the page's fans?
- Are any of the fans for the page particularly outspoken and participatory?
- What are the most common topics being talked about on the page?

Your imagination is the only limitation to what you might be able to ask of the Graph API for a Facebook page when you are mining its content for insights, and these questions should get you headed in the right direction. Along the way, we'll also use these questions as the basis of some comparisons amongst other pages.

Recall that the starting point for our journey might have been a search for "social web" that revealed a book entitled Mining the Social Web per the following search result item:

```
    {
       "category": "Book",
       "name": "Mining the Social Web",
       "id": "146803958708175"
      }
```

Thus, we can take the item's id and use it as the basis of a graph query through `get_object` with an instance of `facebook.GraphAPI`. The code is a quick one-liner that produces the following results:

Example 2-8. Results for a Graph API query for Mining the Social Web

```
# Get an instance of Mining the Social Web
mtsw_id = '146803958708175'
pp(g.get_object(mtsw_id))

{
 "category": "Book",
 "username": "MiningTheSocialWeb",
 "about": "Analyzing Data from Facebook, Twitter, LinkedIn, and Other Social...",
 "talking_about_count": 22,
 "description": "Facebook, Twitter, and LinkedIn generate a tremendous amount...",
 "company_overview": "Like It here on Facebook!\n\nFollow @SocialWebMining...",
 "release_date": "January 2011",
 "can_post": true,
 "cover": {
  "source": "https://sphotos-b.xx.fbcdn.net/...",
  "cover_id": 474206292634605,
```

3. Throughout this section, keep in mind that the responses to these queries reflect data for the 1st Edition of the book, since that's what's available at the time of this writing. Your exact query results may vary somewhat.

```
  "offset_x": -41,
  "offset_y": 0
 },
 "mission": "Teaches you how to...\n\n* Get a straightforward synopsis of the...",
 "name": "Mining the Social Web",
 "founded": "January 2011",
 "website": "http://amzn.to/d1Ci8A",
 "link": "http://www.facebook.com/MiningTheSocialWeb",
 "likes": 911,
 "were_here_count": 0,
 "general_info": "Analyzing Data from Facebook, Twitter, LinkedIn,...",
 "id": "146803958708175",
 "is_published": true
}
```


The interesting analytical results from the query response are the book's `talking_about_count` and `like_count`. The `like_count` is a good indicator of the page's overall popularity, so a reasonable response to a query about the page's overall popularity is that there are 911 Facebook fans for the page, and 22 of them have recently been engaging in discussion. Given that Mining the Social Web is a fairly niche technical book, this seems like a reasonable fan base. For any kind of popularity analysis, however, comparables are essential for understanding the broader context. There are a lot of ways to draw comparisons, but a couple of interesting data points are that the book's publisher, O'Reilly Media has around 34,000 likes, the Python programming language has around 80,000 likes. Thus, the popularity of Mining the Social Web is approaching 3% of the publisher's entire fan base and is just over 1% of the programming language fan base. Clearly, there is a lot of room for this book's popularity to grow even though it's a niche topic.

Although another interesting comparison would have been a niche book similar to Mining the Social Web, it wasn't easily possible to find any good apples-to-apples comparisions by comparing Facebook page data, because at the time of this writing, it isn't possible to search for pages and limit by constraints such as "book" as a category. For example, you can't search for pages and limit the result set to books in order to narrow in on a good comparable; instead, you'd have to search for pages and then filter the result set by category to to retrieve only the books. Still, there are a couple of options to consider:

- Search for another O'Reilly title that you know to be a similar kind of niche book such as Programming Collective Intelligence and see what turns up. The Graph API search results for a query of "Programming Collective Intelligence", do turn up a community page with almost 400 likes, so all things being equal, it's interesting that an almost 7 year old book has almost half the number of likes as Mining the Social Web without an active author maintaining a page for it.
- Take advantage of concepts from Facebook's Open Graph concepts in order to draw a comparison. For example, the O'Reilly catalog contains entries for all of O'Reilly's

titles, implements OGP for these titles, and there is a page (and thus Like buttons) for both Mining the Social Web (http://shop.oreilly.com/product/0636920010203.do) and Programming Collective Intelligence (http://shop.oreilly.com/product/9780596529321.do). We can easily make requests to the Graph API to see what data is available and keep tabs on it by simply querying for these URLs in the browser as follows:

— https://graph.facebook.com/http://shop.oreilly.com/product/0636920010203.do

— https://graph.facebook.com/http://shop.oreilly.com/product/9780596529321.do

In terms of a programmatic query with Python, the URLs are the objects that we are querying (just like the URL for the IMDb entry for The Rock was what we were querying earlier), so in code, we can query these objects as shown in Example 2-9. As a subtle but very important distinction, keep in mind that even though both the O'Reilly catalog page and the Facebook fan page for *Mining the Social Web* logically represent the same book, the node (and corresponding metadata such as the number of likes) that corresponds to the Facebook Page versus the O'Reilly catalog page are completely independent.

Example 2-9. Querying the Graph API for Open Graph objects by their URLs

```
# MTSW catalog link
pp(g.get_object('http://shop.oreilly.com/product/0636920010203.do'))

# PCI catalog link
pp(g.get_object('http://shop.oreilly.com/product/9780596529321.do'))
```

Thus, an "apples-to-oranges" comparison might be the best insight we can glean at the moment, but it's still insightful nonetheless. Hopefully, enhancements to the Graph API as part of the new Graph Search product will facilitate more sophisticated queries as the future unfolds.

As an alternative to analyzing tech books, let's take just a moment to broaden the scope of the discussion to something much more mainstream and see what turns up. The never-ending soft drink war between Coke and Pepsi seems like an innocuous but potentially interesting topic to consider, so let's set out to determine which one is the most popular according to Facebook? As you now know, the answer is just a couple of graph queries away as illustrated in Example 2-10:

Example 2-10. Comparing likes between Coke and Pepsi Fan Pages

```
# Find Pepsi and Coke in search results

pp(g.request('search', {'q' : 'pepsi', 'type' : 'page', 'limit' : 5}))
pp(g.request('search', {'q' : 'coke', 'type' : 'page', 'limit' : 5}))
```

```
# Use the ids to query for likes

pepsi_id = '56381779049'
coke_id = '40796308305'

# A little number formatting help
def int_format(n): return "{:,}".format(n)

print "Pepsi likes:", int_format(g.get_object(pepsi_id)['likes'])
print "Coke likes:", int_format(g.get_object(coke_id)['likes'])
```

The results are somewhat striking:

```
Pepsi likes: 9,677,881
Coke likes: 62,735,664
```

Would you have expected that Coke has almost 7 times the popularity of Pepsi on Facebook? An interesting exercise would be to drill in further and try to determine what might be the cause. For example, are there particular indicators you can find in the data that suggests that Coke launches massive advertising campaigns or does anything particularly interesting to engage users in a way that Pepsi does not? In approaching a question like this one, bear in mind that although there are likely to be interesting indicators in the Facebook data itself, the overall scope is very broad and may also have a number of dependent variables outside of what you might find in Facebook data.

As one possible source of investigation, you might consult stock market information and see if the number of likes correlate at all with the overall market capitalization, which could be an indicator of the overall size of the companies. If you were to look up this information at the time of this writing, however, the results might surprise you in that the market capitalization of Coke (NYSE:KO) is around 178B whereas Pepsi (NYSE:PEP) is 121B. Although analysis of companies at a financial level is a very complex exercise in and of itself, the overall market capitalization of the companies differs by only around 30%, which is a far cry from a 700% difference in Facebook popularity, so it seems reasonable to think that each company probably has similar means available to it and probably sells similar amounts of product. While in some cases, stock market data might have revealed something immediately interesting to consider, nothing surfaced by only considering market capitalization in this brief tangent. Digging deeper into what now seems like a bit of a phenomenon is left as an exercise.

There are practically limitless possibilities for analyzing a Facebook Page, and a great starting point is to analyze the Page's feed, although more specific kinds of filters such as the Page's shared links can be queries for analysis as well. Example 2-11 demonstrates how to query for the page's feed and links to get you started.

The differences between feeds, posts, statuses, etc. can initially be a bit confusing. This Stack Overflow response to the question, What is the difference between feed, posts and statuses in Facebook Graph API? does a good job of concisely clarifying most of the key differences.

Example 2-11. Querying a Page for its "feed" and "links" connections

```
pp(g.get_connections(mtsw_id, 'feed'))
pp(g.get_connections(mtsw_id, 'links'))
```

An interesting place that you might head from here would be to determine which posts in the feed are the most popular as determined by either the number of comments posted or the number of likes. Are any particular kinds of posts more popular than others? For example, are posts with links more popular than posts with photos? What characteristics can you identify that makes a post go viral as opposed to just getting a couple of likes?

Examining Friendships

Let's now use our knowledge of the Graph API to examine the friendships from your own social network. As is always the case, the imagination is the only limitation of what is possible, but here are some interesting questions to get the creative juices flowing include:

- Are there any topics or special interests that are especially pronounced within your social nework?
- Does your social network contain many mutual friendships or even larger cliques?
- How well-connected are the people in your social network?
- Are any of your friends particularly outspoken or passionate about anything you might also be interested in learning more about?

As helpful context in reading this section, bear in mind that the query results and analysis shown in this section is based on real data from the author's own Facebook account that consists of a social network of less than a dozen or so relatively close friends spanning multiple seasons of life. However, there's no reason the same principles and techniques won't apply to your own larger social network.

Let's set out to examine the question about whether or not any topics or special interests exist within your social network and explore from there. A logical starting point for answering this query is to aggregate the likes for each of your friends and try to deter-

mine if there are any particularly high frequency items that appear. Example 2-12 demonstrates how to build a frequency distribution of the likes in your social network as the basis of further anlaysis. Keep in mind that if any of your friends may have privacy settings set to not share certain types of personal information like their likes with apps, you'll often see empty results as opposed to any kind of explicit error message.

Example 2-12. Query for all of your friends' likes

```
friends = g.get_connections("me", "friends")['data']

likes = { friend['name'] : g.get_connections(friend['id'], "likes")['data']
          for friend in friends }
```

There's nothing particularly tricky about collecting your friends' likes and building up a nice data structure although this might be one of your first encounters with a dictionary comprehension. Just like a list comprehension, a dictionary comprehension is iterating over a list of items and collecting values (key/value pairs in this case) that are to be returned.

You may find it interesting to try out the Graph API's new field expansion feature and issue a single query for all of your friends' likes in a a single request. With the `facebook` package, you could do it like this:

```
g.get_object('me',
fields='id,name,friends.fields(id,name,likes)')
```

With a useful data structure called `likes` in hand that contains your friends and their likes, let's start off our analysis by calculating the most popular likes across all of your friends. The `Counter` class provides an easy way to build a frequency distribution that will do just the trick as is illustrated in Example 2-13, and we can use the `prettytable` package to neatly format the results so that they're more readable and easier on the eye.

Example 2-13. Calculating the most popular likes amongst your friends

```
# Analyze all likes from friendships for frequency

from prettytable import PrettyTable
from collections import Counter
friends_likes = Counter([like['name']
                         for friend in likes
                           for like in likes[friend]
                               if like.get('name')])

pt = PrettyTable(field_names=['Name', 'Freq'])
pt.align['Name'], pt.align['Freq'] = 'l', 'r'
[ pt.add_row(fl) for fl in friends_likes.most_common(10) ]
```

```
print 'Top 10 likes amongst friends'
print pt
```

Sample results follow:

```
Top 10 likes amongst friends
+-------------------------+------+
| Name                    | Freq |
+-------------------------+------+
| Crossfit Cool Springs   |    4 |
| CrossFit                |    3 |
| The Pittsburgh Steelers |    3 |
| Working Out             |    3 |
| The Bible               |    3 |
| Skiing                  |    2 |
| Star Trek               |    2 |
| Seinfeld                |    2 |
| Jesus                   |    2 |
+-------------------------+------+
```

It appears that the common theme of exercise/sports is a common theme within this social network with religion/Christianity possibly being a common theme as well. Let's dig a little bit further and analyze the categories of likes that exist within the social network to see if the same themes exist. Example 2-14 illustrates a variation of the previous example that shows how.

Example 2-14. Calculating the most popular categories for likes amongst your friends

```
# Analyze all like categories by frequency

friends_likes_categories = Counter([like['category']
                                    for friend in likes
                                      for like in likes[friend]])

pt = PrettyTable(field_names=['Category', 'Freq'])
pt.align['Category'], pt.align['Freq'] = 'l', 'r'
[ pt.add_row(flc) for flc in friends_likes_categories.most_common(10) ]

print "Top 10 like categories for friends"
print pt
```

Sample results from the query is a tuple with a similar structure as before:

```
Top 10 like categories for friends
+-------------------------+------+
| Category                | Freq |
+-------------------------+------+
| Musician/band           |   62 |
| Book                    |   46 |
| Movie                   |   43 |
| Interest                |   40 |
| Tv show                 |   31 |
| Public figure           |   31 |
```

```
| Local business          |   25 |
| Community               |   24 |
| Non-profit organization |   21 |
| Product/service         |   17 |
+-------------------------+------+
```

There are no explicit mentions of sports or religion, but what is interesting is how much higher the frequencies are on some of these "unexpected" categories such as "Musician/band" or "Book". It may be that there are simply a lot of highly eclectic non-overlapping interests within the social network. Something that may shed further light on the situation and be interesting in and of itself is to calculate how many likes exist for each friend. For example, do most friends have a similar number of likes or is the number of likes highly skewed? Having additional insight into the kind of underlying distribution helps to inform some of the things that may be happening when the data is aggregated. In Example 2-15, we'll calculate a frequency distribution that shows the number of likes for each friend to get an idea of how the categories from the previous example may be skewed.

Example 2-15. Calculating the number of likes for each friend and sorting by frequency

```
# Build a frequency distribution of number of likes by
# friend with a dictionary comprehension and sort it in
# descending order

from operator import itemgetter

num_likes_by_friend = { friend : len(likes[friend])
                        for friend in likes }


pt = PrettyTable(field_names=['Friend', 'Num Likes'])
pt.align['Friend'], pt.align['Num Likes'] = 'l', 'r'
[ pt.add_row(nlbf)
  for nlbf in sorted(num_likes_by_friend.items(),
                     key=itemgetter(1),
                     reverse=True) ]

print "Number of likes per friend"
print pt
```

Sample results have the familar form of a tuple with a friend and frequency value. Some results (sanitized of last names) follow:

```
Number of likes per friend
+--------------------+-----------+
| Friend             | Num Likes |
+--------------------+-----------+
| Joshua             |       187 |
| Derek              |       146 |
| Heather            |        84 |
| Rick               |        69 |
```

```
| Patrick            |        42 |
| Bryan              |        38 |
| Ray                |        17 |
| Jamie              |        14 |
| Bas                |         0 |
+--------------------+-----------+
```

The more time you spend really trying to understand the data, the more insights you'll glean, and by now, you are hopefully starting to get a more holistic picture of what's happening. We now know that the distribution of likes across the data is enormously skewed across a small number of friends and that any one friend's results could be highly contributring to the results that break down the frequencies of category for each like. There are a number of interesting directions that we could go at this point. One possibility would be to start to compare smaller samples of friends for some kind of similarity or further analyze likes. For example, does Joshua account for 90% of the liked TV shows? Does Derek account for the most significant majority of liked music? The answers to these questions are well within your grasp at this point to answer.

Instead, however, let's ask another question that's generally relevant and interesting: which friends are most similar to the ego[4] in the social network? In order make any kind of interesting similarity comparison between two things, we'll need a similarity function. The simplest possibility is likely to be one of the best starting points, so let's start out with "number of shared likes" to compute similarity between the ego and frienships. All that needs to happen in order to compute the similarity between the ego of the network and the friendships is the ego's likes and some help from the `set` object's intersection operator, which makes it possible to take two lists of items and compute the overlapping items from each of them. Example 2-16 illustrates how to compute the overlapping likes between the ego and friendships in the network as the first step in finding the most similar friends in the network.

Example 2-16. Finding common likes between an ego and its friendships in a social network

```
# Which of your likes are in common with which friends?
my_likes = [ like['name']
             for like in g.get_connections("me", "likes")['data'] ]

pt = PrettyTable(field_names=["Name"])
pt.align = 'l'
[ pt.add_row((ml,)) for ml in my_likes ]
print "My likes"
print pt

# Use the set intersection to find common likes
```

4. The ego of a social network is is logical center or basis. In this case, the ego of the network is the author of this book -- the person whose social network we are examining.

```
common_likes = list(set(my_likes) & set(friends_likes))

pt = PrettyTable(field_names=["Name"])
pt.align = 'l'
[ pt.add_row((cl,)) for cl in common_likes ]
print
print "My common likes with friends"
print pt
```

Abbreviated output containing the results of the overlapping likes that are in common for this social network follow:

```
My likes
+-----------------------------------------------------------------------------------------------------+
| Name                                                                                                |
+-----------------------------------------------------------------------------------------------------+
| Snatch (weightlifting)                                                                              |
| First Blood                                                                                         |
| Robinson Crusoe                                                                                     |
| The Godfather                                                                                       |
| The Godfather                                                                                       |
                                                    ...
| The Art of Manliness                                                                                |
| USA Triathlon                                                                                       |
| CrossFit                                                                                            |
| Mining the Social Web                                                                               |
+-----------------------------------------------------------------------------------------------------+

My common likes with friends
+------------------------------+
| Name                         |
+------------------------------+
| www.SEALFIT.com              |
| Rich Froning Jr. Fan Site    |
| CrossFit                     |
| The Great Courses            |
| The Art of Manliness         |
| Dan Carlin - Hardcore History |
| Mining the Social Web        |
| Crossfit Cool Springs        |
| Maranatha Koinōnía           |
+------------------------------+
```

Coming back full fircle, it's perhaps not too surprising that the common theme of sports/exercise one again emerges (but with additional detail this time) as do some topics related to Christian religion[5]. There many more interesting questions to ask (and answer), but let's taper off this section by completing the second half of this particular

5. Maranatha Koinōnía is a Chriatian church (where koinōnía is the Greek word for "fellowship") and Rich Froning Jr. is a well-known CrossFit athlete who is also publicly known as a Christian.

query, which is to now find the particular friends that share the common interests with the ego in the network. Example 2-17 shows how to do this by iterating over the friendships with a double list comprehension:

Example 2-17. Calculating the most similar friends to an ego in a social network

```
# Which of your friends like things that you like?

similar_friends = [ (friend, friend_like['name'])
                     for friend, friend_likes in likes.items()
                       for friend_like in friend_likes
                         if friend_like.get('name') in common_likes ]


# Filter out any possible duplicates that could occur

ranked_friends = Counter([ friend for (friend, like) in list(set(similar_friends)) ])


pt = PrettyTable(field_names=["Friend", "Common Likes"])
pt.align["Friend"], pt.align["Common Likes"] = 'l', 'r'
[ pt.add_row(rf)
  for rf in sorted(ranked_friends.items(),
                   key=itemgetter(1),
                   reverse=True) ]
print "My similar friends (ranked)"
print pt
```

By now, you should be familiar with the processing; all that we've done is iterated over the variables we've built up so far to build a list of expanded out tuples of the form *(friend, friend's like)* and then used it to compute a frequency distribution to determine which friends have the most common likes. Sample results for this query follow:

```
My similar friends (ranked)
+----------+--------------+
| Friend   | Common Likes |
+----------+--------------+
| Derek    |            4 |
| Jamie    |            3 |
| Joshua   |            2 |
| Heather  |            2 |
| Patrick  |            1 |
+----------+--------------+
```

As you are probably thinking, there is an abundance of interesting questions that can be investigated with even a small sliver of of data from your Facebook friends. We've just scratched the surface, but hopefully, these exercises of have been helpful in terms of framing up some good starting points that can be further explored. It doesn't take very much imagination to continue down this road or to pick up with a different angle and start down an entirely different one. To illustrate just one possibility, let's spend just

a moment and introduce a nifty way to visualize some of your Facebook friends data that's along a different line of thinking before closing out this chapter.

[XXX: Anonymize and provide data from this section to readers so that they can follow along with zero deviations or unexpected hiccups?]

Constructing and Analyzing Mutual Friendship Graphs

Unlike Twitter, which is an inherently open network in which you could crawl "friendships" over an extended period of time and build a large graph for any given starting point, Facebook data is much richer and laden with personally identifiable and sensitive properties about people, so the privacy and access controls make it much more closed. While you can use the Graph API to access data for the authenticating user and the authenticating user's friends, you cannot access data for arbitrary users beyond those boundaries unless it is exposed as publicly available. One Graph API operation that is quite interesting is the ability to get the mutual frienships (available through the mutualfriends API and documented as part of the User object) that exist within your social network (or the social network for the authenticating user)? In other words, which of your friends are also friends with one another? From a graph analytics perspective, analysis of an ego graph for mutual friendships can very naturally be formulated as a clique detection problem.

For example, if Abe is friends with Bob, Carol, and Dale, and Bob and Carol are also friends, the largest ("maximum") clique in the graph exists among Abe, Bob, and Carol. If Abe, Bob, Carol, and Dale were all mutual friends, however, the graph would be fully connected, and the maximum clique would be of size 4. Adding nodes to the graph might create additional cliques, but it would not necessarily affect the size of the maximum clique in the graph. In the context of the social web, cliques are fascinating because they are representative of mutual friendships, and the maximum clique is interesting because it indicates the largest set of common friendships in the graph. Given two social networks, comparing the sizes of the maximum friendship cliques might provide a good starting point for analysis about various aspects of group dynamics such as teamwork, trust, productivity, etc. Figure 2-6 illustrates a sample graph with the maximum clique highlighted. This graph would be said to have a *clique number* of size 4.

Technically speaking, there is a subtle difference between a *maximal* clique and a *maximum* clique. The maximum clique is the largest clique in the graph (or cliques in the graph, if they have the same size). A maximal clique, on the other hand, is one that is not a subgraph of another clique. The maximum clique is also a maximal clique in that it isn't a subgraph of any other clique; however, various other maximal cliques often exist in graphs and need not necessarily share any nodes with the maximum clique. Figure 2-6, for example, illustrates a maximum clique of size 4, but there are several other maximal cliques of size 3 in the graph as well.

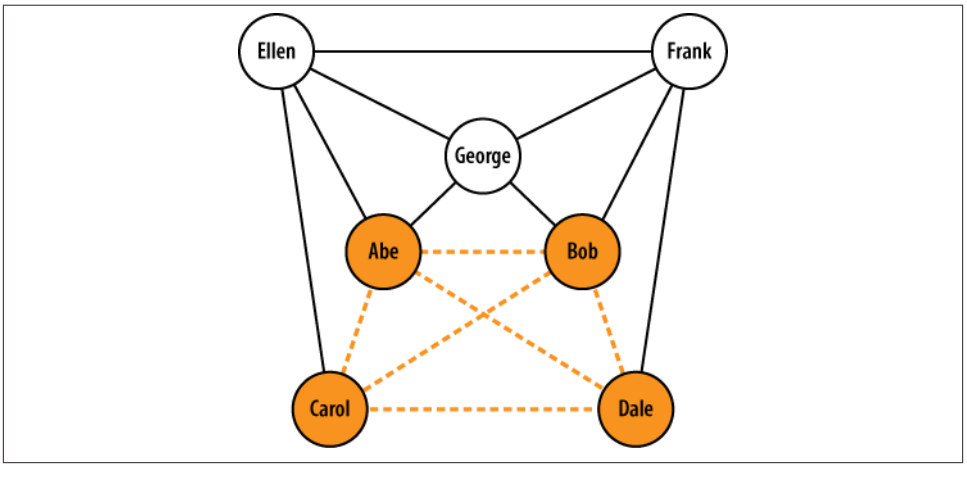


Figure 2-6. An example graph containing a maximum clique of size 4

Finding cliques is an NP-complete problem (implying an exponential runtime), but there is an amazing Python package called `networkx` (pronounced either "networks" or "network x") that provides extensive graph analytics functionality, including a `find_cliques` method that delivers a solid implementation of this hard problem. Just be advised that it might take a long time to run as graphs get beyond a reasonably small size, so runtimes may vary. Example 2-18 and Example 2-19 demonstrate how to use Facebook data to construct a graph of mutual friendships and then use NetworkX to analyze the cliques within the graph.

Example 2-18. Construct a graph of mutual friendships

```
import networkx as nx # pip install networkx
import requests # pip install requests

friends = [ (friend['id'], friend['name'],)
               for friend in g.get_connections('me', 'friends')['data'] ]
```

```
url = 'https://graph.facebook.com/me/mutualfriends/%s?access_token=%s'

mutual_friends = {}

# This loop spawns a separate request for each iteration, so
# it may take a while. Optimization with a thread pool or similar
# technique would be possible
for friend_id, friend_name in friends:
    r = requests.get(url % (friend_id, ACCESS_TOKEN,) )
    response_data = json.loads(r.content)['data']
    mutual_friends[friend_name] = [ data['name']
                                    for data in response_data ]

nxg = nx.Graph()

[ nxg.add_edge('me', mf) for mf in mutual_friends ]

[ nxg.add_edge(f1, f2)
  for f1 in mutual_friends
      for f2 in mutual_friends[f1] ]
```

Example 2-19. Find and analyze cliques in a graph of mutual friendships

```
# Finding cliques is a hard problem, so this could
# take awhile for large graphs.
# See http://en.wikipedia.org/wiki/NP-complete and
# http://en.wikipedia.org/wiki/Clique_problem

cliques = [c for c in nx.find_cliques(nxg)]

num_cliques = len(cliques)

clique_sizes = [len(c) for c in cliques]
max_clique_size = max(clique_sizes)
avg_clique_size = sum(clique_sizes) / num_cliques

max_cliques = [c for c in cliques if len(c) == max_clique_size]

num_max_cliques = len(max_cliques)

max_clique_sets = [set(c) for c in max_cliques]
friends_in_all_max_cliques = list(reduce(lambda x, y: x.intersection(y),
                                  max_clique_sets))

print 'Num cliques:', num_cliques
print 'Avg clique size:', avg_clique_size
print 'Max clique size:', max_clique_size
print 'Num max cliques:', num_max_cliques
print
print 'Friends in all max cliques:'
print json.dumps(friends_in_all_max_cliques, indent=1)
print
```

```
print 'Max cliques:'
print json.dumps(max_cliques, indent=1)
```

In the 1st Edition of Mining the Social Web (2011), analysis Tim O'Reilly's 600+ Twitter friendships revealed some interesting insights. At the time, there were over 750,000 total cliques in the network, with an average clique size of 14 and a maximum clique size of 26. In other words, the largest number of people who are fully connected among Tim's friends was 26, and there were 6 distinct cliques of this size. The same individuals accounted for 20/26 of the population in all 6 of those cliques. In terms of insight, you might think of these members as being especially foundational for Tim's Twitter network.

Sample output for Example 2-19 follows and illustrates that for the 10 total people in the social network, there are 4 cliques of size 4 with the ego ("me") and one other person being common to all of them. Although the other person in common to all of the cliques is not guaranteed to be the 2nd most highly connected person in the network, this person is likely to be among the most influential because of the relationships in common. What other insights can you discover by analyzing the graph and some of the other algorithms that NetworkX provides?

```
Num cliques: 6
Avg clique size: 3
Max clique size: 4
Num max cliques: 4

Friends in all max cliques:
[
 "me",
 "Bas"
]

Max cliques:
[
 [
  "me",
  "Bas",
  "Joshua",
  "Heather"
 ],
 [
  "me",
  "Bas",
  "Ray",
  "Patrick"
 ],
 [
  "me",
```

```
    "Bas",
    "Ray",
    "Rick"
   ],
   [
    "me",
    "Bas",
    "Jamie",
    "Heather"
   ]
  ]
```

Example 2-19 could be modified in any number of ways to compute interesting things, and clearly, there's much more we could do than just detect the cliques. Plotting the locations of people involved in cliques on a map to see whether there's any correlation between tightly connected networks of people and geographic locale, and analyzing information in their profile data and content in poasts might be a couple of good starting points. In the next section, we'll learn how to put together a concise but effective visualization of mutual friendships in an intuitive graphical format.

Visualizing Mutual Friendships with Force Directed Graphs

D3.js is a truly state-of-the-art JavaScript toolkit that can render some beautiful visualizations in the browser with an intuitive approach that involves manipulating objects with a series of data-driven transformations. If you haven't already encountered D3, then you really should take a few moments to browse the example gallery to get a feel for what is possible. You will be impressed.

A tutorial of how to use D3 is well outside the scope of this book, and there are numerous tutorials and discussions online about how to use many of its exciting visualizations. What we'll do in this section before rounding out the chapter is render an interactive visualization for the mutual friendship graph we are currently working with as introduced in the previous section. Abstractly, a graph is just a mathematical construct and doesn't have a visual representation, but a number of layout algorithms are available that can render the graph in two-dimensional space so that it displays rather nicely (although you may need to tweak some of the layout parameters from time to time to get things just right.)

NetworkX can emit a format that is directly consumable by D3, so very little work is necessary to visualize the graph beyond staging a web server to host a web page that renders it. The HTML that embeds the necessary style and scripts is included with the IPython Notebook for this chapter in a folder called *viz*. Example 2-20 demonstrates how to serialize out the graph for rendering, and Example 2-21 uses Flask, the same embedded web server that we used earlier to handle the OAuth direct, to open up your web browser and serve up a web page displaying an interactive graph such as the one shown in Figure 2-7.

Example 2-20. Serialize a NetworkX graph to a file for consumption by D3

```
from networkx.readwrite import json_graph

nld = json_graph.node_link_data(nxg)

json.dump(nld, open('viz/force.json','w'))
```

Example 2-21. Visualize a mutual friendship graph with D3

```
import os
import webbrowser
from flask import Flask
from threading import Timer

# Configure an embedded web server to serve static content
# in the 'viz' folder of the working directory
webserver = Flask("MutualFriendships", static_folder=os.getcwd() + "/viz")

# Prevent a race condition by waiting to open the web browser until
# the embedded web server is running
Timer(1, webbrowser.open, args=['http://localhost:5000/viz/force.html']).start()

webserver.run() # Select"Kernel => Interrupt" from the menu to stop the web server...
```

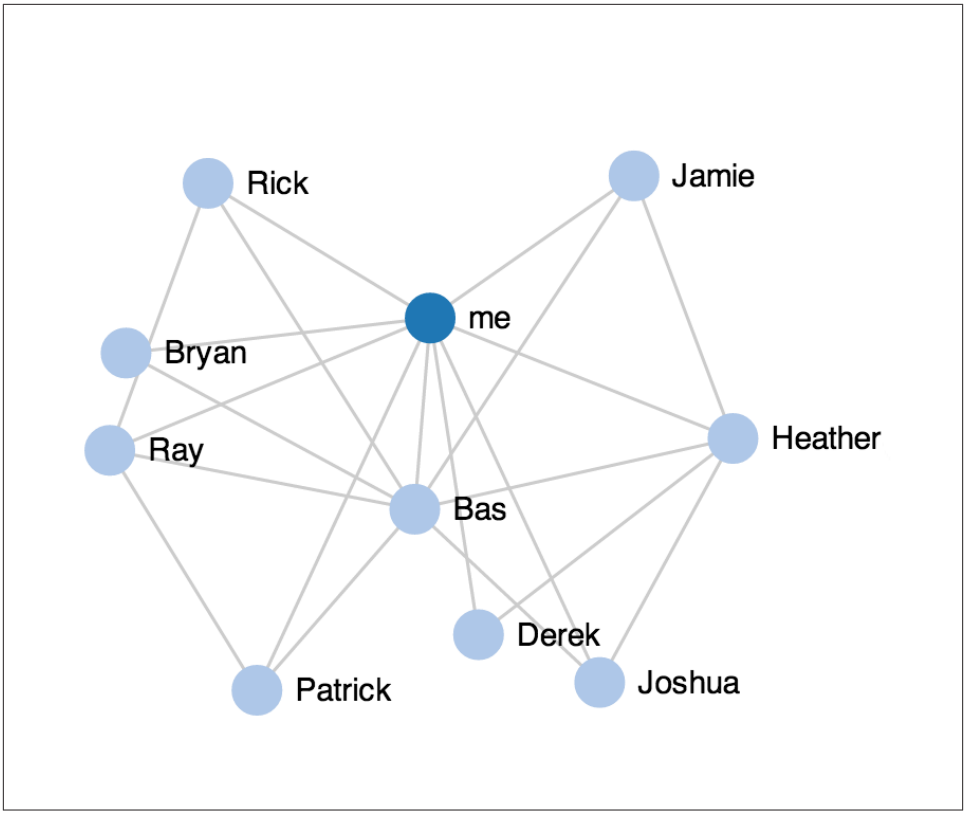


Figure 2-7. A graph of mutual friendships within a Facebook social network rendered by D3

Closing Remarks

The goal of this chapter was to teach you about the Graph API, how Open Graph protocol can create connections between arbitrary web pages and Facebook's Social Graph, and how to programatically query the Social graph to discover insight about Facebook pages and your own social network. If you've worked through the examples in this chapter, you should have little to no trouble probing the Social Graph for answers to questions that may have very valuable answers. Keep in mind that as you explore a data set, not to mention a data set as enormous and interesting as Facebook's Social Graph, that you really just need a good starting point. As you investigate answers to an initial query, you'll likely follow a very natural course of exploration that will successively refine your understanding of the data and get you closer to the answers that you are looking for.

The possibilities for mining data on Facebook are immense, but be respectful of privacy, and always comply with Facebook's terms of service to the best of your ability. Unlike data from Twitter and some other sources that are inherently more open in nature, Facebook data can be quite sensitive in nature, especially if you are analyzing your own social network. Hopefully, this chapter has made it apparent why it's so little wonder that there are so many exciting possibilities for what can be done with social data and the enormous amount of value that is tucked away on Facebook.

Recommended Exercises

- Analyze data from the fan page for something you're interested in on Facebook and attempt to analyze the natural language in the comments stream to gain insights. What are the most common topics being discussed? Can you tell if fans are particularly happy or upset about anything?
- Select two different fan pages that are similar in nature and compare/contrast them. For example, what similarities and differences can you identify between fans for Chipotle Mexican Grill and Taco Bell? Can you find anything surprising?
- Analyze your own friendships and try to determine if your own network has any natural rallying points or common interests? What is the glue that binds your network together?
- The number of Facebook objects available to the Graph API is enormous. Can you examine objects such as photos or checkins to discover insights about anyone in your network? For example, who posts the most pictures and what are they about based on the comments stream? Where do your friends checkin most often?
- Use histograms as introduced toward the end of Chapter 1 to further slice and dice your friends' likes data.
- Use the Graph API to collect other kinds of data and find a suitable D3 visualization for rendering it. For example, can you plot where your friends live or where they grew up on a map? Which of your friends still live in their hometowns?
- Harvest some Twitter data, construct a graph, and analyze/visualize it using the same techniques as introduced in this chapter.
- Try out some different similarity metrics to compute your most similar friendships. The Jaccard Index is a good starting point. [XXX: xref section in Google+ chapter with additional similarity metrics when linked into book.]

CHAPTER 3

Mining Your LinkedIn Professional Network: Faceting Job Titles, Clustering Colleagues, and More

This chapter introduces techniques and considerations for mining the troves of data tucked away at LinkedIn, a popular and powerful social networking site focused on professional and business relationships. Although LinkedIn may initially seem like any other social network, the data its API provides is inherently of a very different nature. If you likened Twitter to a busy public forum with everyone talking at once as part of a free-for-all and Facebook to a very large room filled with all of your friends and family as big get-together with conversation flowing about things that are (mostly) appropriate for dinner conversation, then you might liken LinkedIn to a private event with a semi-formal dress code where everyone is on their very best behavior and generally trying to ensure that everyone else in the room understands the specific value and expertise that they could provide in the pursuit of economic opportunities.

Given the somewhat sensitive nature of the data that's tucked away at LinkedIn, its API has its own subtle nuances that make it a bit different from many of the others we've looked at in this book, because people who join LinkedIn are principally interested in the business opportunities that it provides as opposed to arbitrary socializing and will necessarily be providing sensitive details about business relationships, job histories, etc. For example, while you can generally access all of the interesting details about your LinkedIn connections' educational histories, previous work positions, etc. just as you could acceess this same kind of information for your friends on Facebook, you cannot determine whether two arbitrary people are "mutually connected" as you could with Facebook, and the details of . The absence of such an API method is intentional, and by design, the API doesn't lend itself to being modeled as a social graph such as Facebook or Twitter, therefore requiring that you ask different types of questions about the data that's available to you.

The remainder of this chapter gets you setup to access data with the LinkedIn API and introduces some fundamental data mining techniques that can help you cluster colleagues according to a similarity measurement in order to answer the following kinds of queries:

- Which of your connections are the most similar based upon a criterion like job title?
- Which of your connections have worked in companies you want to work for?
- Where do most of your connections reside geographically?

In all cases, the pattern for analysis with a clustering technique is essentially the same: extract some features from data in a colleague's profile, define a similarity measurement to compare the features from each profile, and use a clustering technique to group together colleagues that "similar enough." The approach works well for LinkedIn data, and as a general approach, you can apply these same techniques to just about any kind of other data that you'll ever encounter.

Clustering is a fundamental data mining technique, and as part of a proper introduction to it, this chapter includes some footnotes and interlaced discussion of a mathematical nature that undergirds the problem. Although you should strive to eventually understand these details, you don't need to understand all of the finer points to successfully employ clustering techniques, and you certainly shouldn't feel any pressure to understand them the first time that you encounter them. It may take a little bit of reflection for some of the discussion to digest, especially if you don't have a mathematical background.

The LinkedIn Developer Network

Like most of the other chapters in this book, you'll need a LinkedIn account and a few colleagues in your professional network to follow along with the examples. If you don't have a LinkedIn account, you can still apply many of the techniques we'll explore to other domains that provide a means of exporting information similar to what you'd find in an address book, but it just won't be quite as engaging as you read along with this chapter, so go ahead and start developing a professional network if you don't already have one as a worthwhile investment in your professional life.

Once you have a LinkedIn account, the next step is to access (and bookmark) the LinkedIn Developers site. As with Twitter and Facebook in earlier chapters, LinkedIn uses OAuth so that users can authorize applications to access their account data without having to share a secret like a password, which would be a very bad idea. Once you've reached the developer site, you'll be able to create a sample application from https://

www.linkedin.com/secure/developer. Take note of your application's "API Key" and "Secret Key" that we'll use to access to the API momentarily. Figure 3-1 illustrates the form that you'll see once you have created an application and have access to these credentials. Before moving on, you should take a moment read about what LinkedIn API operations are available to you as a developer by browsing its REST documentation that provides a broad overview of what is possible. Although we'll be accessing the API through a Python package that abstracts the HTTP requests that are involved, the core API documentation is your definitive reference.

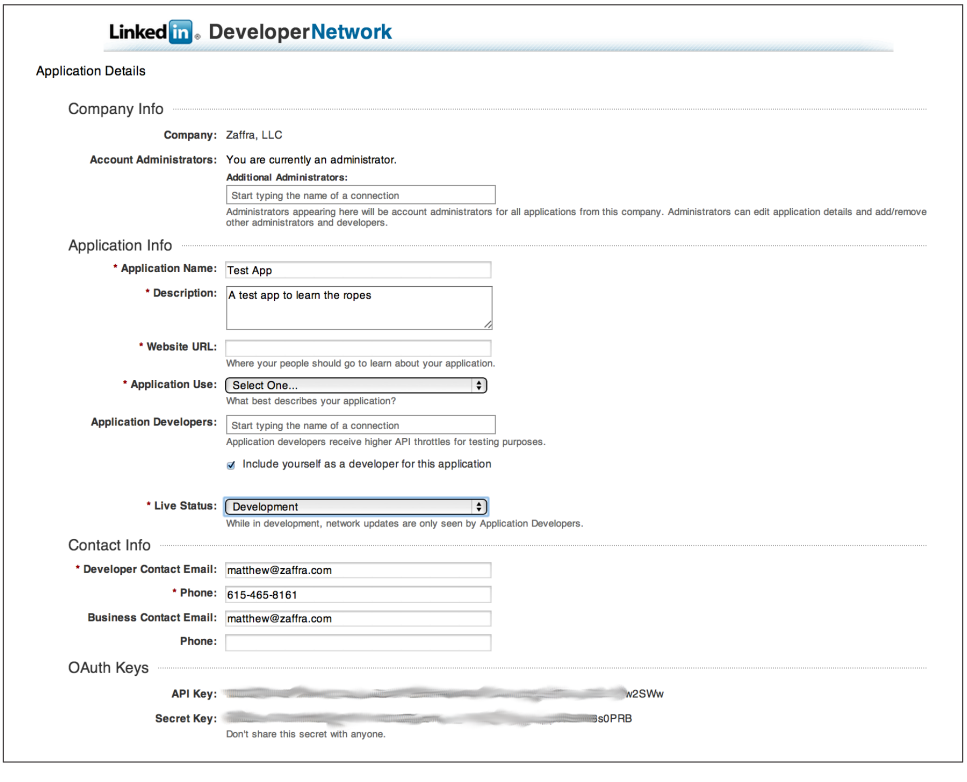


Figure 3-1. To access the LinkedIn API, create an application and take note of your "API Key" and "Secret Key" from the details for your application.

Making LinkedIn API Requests

With the credentials in hand, the overall process is similar to that of any other social network that requires you to do the OAuth dance to access APIs although most of the messy details will be abstracted away since we'll be using the `python-linkedin` package to do most of the tedious work for us. Just `pip install python-linkedin requests` to install it and its dependency on `requests`, a package that trivializes the work involved

in making HTTP requests. Finally, we'll be using Flask to host an embedded web server and handle the OAuth redirect using the same pattern that we applied in Chapter 2, so `pip install flask` if you haven't done so already.

Example 3-1 illustrates a sample script that uses your LinkedIn OAuth credentials to do the OAuth dance and procure an access token, which provides the ultimate means of authorization to access your LinkedIn data.

Example 3-1. Using LinkedIn OAuth credentials to receive an access token an authorize an application

```
import webbrowser
from threading import Timer
from flask import Flask, request
from linkedin import linkedin

# XXX: Get these values from your application's settings for the OAuth flow

ACCESS_KEY = ''
SECRET_KEY = ''

# This value is where LinkedIn will redirect. We'll configure an embedded
# web server to be serving requests here

RETURN_URL = 'http://localhost:5000/oauth_helper'

# Configure an emedded web server that accepts one request, stores a file
# that will need to be accessed outside of the request context, and
# immediately shuts itself down

webserver = Flask("OAuthHelper")
@webserver.route("/oauth_helper")
def oauth_helper():
    code = request.args.get('code')
    f = open('/tmp/flask.authorization_code', 'w') # Store the code as a file
    f.write(code)
    f.close()
    shutdown_after_request = request.environ.get('werkzeug.server.shutdown')
    shutdown_after_request()
    return """<p>Handled redirect and extracted code <strong>%s</strong>
                for authorization</p>
              <p>Close this browser window</p>""" % (code,)

# Send an OAuth request to LinkedIn, handle the redirect, and display the access
# token that's included in the redirect for the user to copy and paste

auth = linkedin.LinkedInAuthentication(ACCESS_KEY, SECRET_KEY, RETURN_URL,
                                       linkedin.PERMISSIONS.enums.values())

app = linkedin.LinkedInApplication(auth)

# Open a browser after a 1 second delay to prevent a possible race condition
```

```
# and ensure that the web server is running and can handle the request

Timer(1, webbrowser.open, args=[auth.authorization_url]).start()

# Run the server to accept the redirect back from LinkedIn and capture the access
# token. This command blocks, but the web server is configured to shut itself down
# after it serves a request, so after the redirect occurs, program flow will continue

webserver.run()

# As soon as the request finishes, the web server shuts down and these remaining commands
# are executed, which exchange an authorization code for an access token. This process
# seems to need full automation because the authorization code expires very quickly.

app.authentication.authorization_code = open('/tmp/flask.authorization_code').read()
app.authentication.get_access_token()

# How you can use the application to access the LinkedIn API...

print app.get_profile()
```

With an OAuth access token in hand, let's briefly explore some of the possible queries that can return data. The last line of Example 3-1 illustrated how to get your basic profile information (as the user who authorized the application.) In short, the calls available to you through an instance of `LinkedInApplication` are the same as what is available through the REST API, and the python-linkedin documentation on GitHub project provides a number of queries to get you started. A couple of APIs of particular interest are the Connections API and the Search API. Recalling that you cannot get "friends of friends" ("connections of connections" in LinkedIn parlance), the Connections API returns a list of your connections, which provides a jumping off point for obtaining profile information. The Search API provides a means of querying for people, companies, or jobs that are available on LinkedIn. Additional APIs are available, and it's worth your while to take a moment and familiarize yourself with what's available. The quality of the data available about your professional network is quite remarkable as it can potentially contain full job histories, company details, geographic information about where positions were located, etc.

As a concrete starting point, the script illustrated in Example 3-2 shows you how to use `app`, an instance of your `LinkedInApplication`, to retrieve extended profile information for your connections who haven't opted out of LinkedIn API access (in which case their first and last name will appear as "private" and additional details will not be available.)

Example 3-2. Retrieving extended profile information for your LinkedIn connections

```
from prettytable import PrettyTable

pt = PrettyTable(field_names=['Name', 'Location'])
pt.align = 'l'
```

```
[ pt.add_row((c['firstName'] + ' ' + c['lastName'], c['location']['name']))
  for c in connections['values']
      if c.has_key('location')]

print pt
```

Sample (anonymized) results follow and display your connections and where they are currently located according to their profiles.

```
+-----------------------------------+-----------------------------------------------+
| Name                              | Location                                      |
+-----------------------------------+-----------------------------------------------+
| Laurel A.                         | Greater Boston Area                           |
| Eve A.                            | Greater Chicago Area                          |
| Jim A.                            | Washington D.C. Metro Area                    |
| Tom A.                            | San Francisco Bay Area                        |
|   ...                             |          ...                                  |
+-----------------------------------+-----------------------------------------------+
```

A full scan of the profile information returned from the Connections API reveals that it's pretty spartan, but you can use field selectors as documented in the Profile Fields online documentation to retrieve additional details, if available. For example, Example 3-3 shows how to fetch a connection's job positions:

Example 3-3. Displaying job position history for your profile and a connection's profile

```
import json

# See http://developer.linkedin.com/documents/profile-fields#fullprofile
# for details on additional field selectors that can be passed in for
# retrieving additional profile information

# Display your own positions...

my_positions = app.get_profile(selectors=['positions'])
print json.dumps(my_positions, indent=1)

# Display positions for someone in your network...

# Get an id for a connection. We'll just pick the first one
connection_id = connections['values'][0]['id']
connection_positions = app.get_profile(member_id=connection_id,
                                       selectors=['positions'])
print json.dumps(connection_positions, indent=1)
```

Sample output reveals a number of interesting details, including, the company name, industry, summary of efforts at each position, and employment dates.

```
{
 "positions": {
  "_total": 10,
  "values": [
```

```
      {
       "startDate": {
        "year": 2013,
        "month": 2
       },
       "title": "Chief Technology Officer",
       "company": {
        "industry": "Computer Software",
        "name": "Digital Reasoning Systems"
       },
       "summary": "I lead strategic technology efforts...",
       "isCurrent": true,
       "id": 370675000
      },
      {
       "startDate": {
        "year": 2009,
        "month": 10
       }
       ...
     ]
    }
   }
```

As might be expected, each API response may not necessarily contain all of the information that you want to know, and some responses may contain more information than you care to receive back. Instead of making multiple API calls to piece together information or potentially stripping out information you don't want to keep, you could take advantage of the field selector syntax that's available to customize the response details. Example 3-4 shows how you could retrieve only the `name`, `industry`, and `id` fields for companies as part of a response for profile positions.

Example 3-4. Using field selector syntax to request additional details for APIs

```
# See http://developer.linkedin.com/documents/understanding-field-selectors
# for more information on the field selector syntax

my_positions = app.get_profile(selectors=['positions:(company:(name,industry,id))'])
print json.dumps(my_positions, indent=1)
```

Once you're familar with the basic API that are available to you, have a few handy pieces of documentation bookmarked, and have made a few API calls to familiarize yourself with the basics, that's about all you need to know to be up and running with LinkedIn. One other important consideration worth noting, however, is LinkedIn's rate-throttling limits that are similar to what you may have experienced with Twitter's API. Be careful when tinkering around with LinkedIn's API because the rate limits don't reset until midnight UTC, and one buggy loop could potentially blow your plans for the next 24 hours if you aren't careful.

With a basic understanding of how to access LinkedIn's API in place, let's dig into some more specific analysis. The next section introduces the concept of a clustering, an unsupervised machine learning technique that is a staple in any data mining toolkit and a tool that you'll definitely want to have in your belt for a variety of data.

Prerequisite Knowledge for Clustering

Clustering[1] which is a very well-studied (but still difficult and messy) problem that's ubiquitous in nearly any industry. is a data mining technique that takes a collection of items and partitions them into smaller collections (clusters) according to some heuristic that is usually designed to compare the similarity of items in the collection. For example, if you were considering a geographic relocation, you might find utility in clustering your LinkedIn connections into some number of geographic regions in order to make a better decision with the idea being that you might have more economic opportunity available if you lived near the centroid of a cluster, near a good source of public transportation within in the cluster, etc. We'll revisit this concept momentarily, but let's take a moment to briefly discuss some nuances associated with clustering. Given the richness of LinkedIn data, being able to answer a number of useful queries about your professional network with clustering; however, in implementing solutions to problems that lend themselves to clustering on LinkedIn or elsewhere, there are at least three common themes you'll encounter over and over again as part of a clustering analysis:

Data normalization

Even when retrieving data from a nice API, it's usually not the case that the data will be provided to you in exactly the format you'd like, but it often takes a lot more than a little bit of munging to get the data into a form suitable for analysis. For example, LinkedIn members can enter in text that describes their job title, so it won't always be the case that you end up with perfectly normalized job titles. One executive might choose to describe their title as "Chief Technology Officer" while another may opt for the more ambiguous "CTO", and still others may choose other variations of the same role. We'll revisit the data normalization problem and implement a pattern for handling certain aspects of it for LinkedIn data momentarily.

Similarity Computation

Assuming you have reasonably well normalized items, you'll need to measure similarity between any two of them, whether they're job titles, company names, professional interests, geographic labels, or any other field you can enter in as variable free text, so you'll need to define a heuristic that can approximate the similarity between any two values. In some situations, computing a similarity heuristic can be quite obvious, but in others, it can be very tricky. For example, comparing the

1. Also commonly referred to as approximate matching, fuzzy matching, and/or deduplication, among many other things.

combined years of career experience for two people might be as simple as some addition operations, but comparing a broad professional element such as "leadership aptitude" in a fully automated manner could be quite a challenge.

Dimensionality Reduction
: In order to cluster all of the items in a set using a similarity metric, it would be ideal to compare every member to every other member. Thus, for a set of n members in a collection, you would perform somewhere on the order of n^2 similarity computations in your algorithm for the worst-case scenario. Computer scientists call this predicament an *n-squared problem* and generally use the nomenclature $O(n^2)$ to describe it;[2] conversationally, you'd say it's a "Big-O of n-squared" problem. However, $O(n^2)$ problems become intractable for very large values of n. Most of the time, the use of the term "intractable" means you'd have to "too long" for a solution to be computed. "Too long" might be minutes, years, or eons, depending on the nature of the problem and its constraints.

Techniques for clustering are a fundamental part of any legitimate data miner's tool belt, because in nearly any sector of any industry—ranging from defense intelligence to fraud detection at a bank to local landscaping companies—there can be a truly immense amount of semi-standardized relational data that needs to be analyzed, and the rise of data scientist job opportunities over the previous years have been a testament to these opportunities. What generally happens is that a company establishes a database for collecting some kind of information, but not every field is enumerated into some predefined universe of valid answers. Whether it's because the application's user interface logic wasn't designed properly, because some fields just don't lend themselves to having static predetermined values, or because it was critical to the user experience that users be allowed to enter whatever they'd like into a text box, the result is always the same: you eventually end up with a lot of semistandardized data, or "dirty records." While there might be a total of N distinct string values for a particular field, some number of these string values will actually relate the same concept. Duplicates can occur for various reasons: erroneous misspellings, abbreviations or shorthand, differences in the case of words, etc.

Although it may not be obvious, this is exactly one of the interesting situations we're faced with in mining LinkedIn data: LinkedIn members are able to enter in their professional information as free text, which results in a certain amount of unavoidable variation. For example, if you wanted to examine your professional network and try to

2. Technically speaking, it is sometimes more precise to describe the time complexity of a problem as $\Theta(n^2)$, because $O(n^2)$ represents a worst case, whereas $\Theta(n^2)$ represents a tighter bound for the expected performance. In other words, $\Theta(n^2)$ says that it's not just an n^2 problem in the worst case, it's an n^2 problem in the best case, too. However, our objective in this chapter isn't to analyze the runtime complexity of algorithms (a difficult subject in and of itself), and we'll stick with the more common $O(n^2)$ notation and describing the worst case.

determine where most of your connections work, you'd need to consider common variations in company names. Even the simplest of company names have a few common variations you'll almost certainly encounter. For example, it should be obvious to most people that "Google" is an abbreviated form of "Google, Inc.", but even these kinds of simple variations in naming conventions must be explicitly accounted for during standardization efforts. In standardizing company names, a good starting point is to first consider suffixes such as ", Inc.", ", LLC", etc.

Normalizing LinkedIn Data

As a necessary and helpful interlude towards a working knowledge of clustering, let's explore a few of the common situations you may face in normalizing LinkedIn data. In this section, we'll implement a common pattern for normalizing company names and job titles. As a more advanced exercise, we'll briefly discuss the problem of disambiguating and geocoding geographic references from LinkedIn profile information. In other words, we'll attempt to convert labels from LinkedIn profiles such as "Greater Nashville Area" to coordinates that can be plotted on a map.

Normalizing and Counting Companies

Let's take a stab at standardizing company names from your professional network. The two primary ways you can access your LinkedIn data are by either using the LinkedIn API to programmatically retrieve the relevant fields or by a slightly lesser known mechanism that allows you to export your professional network as address book data, which includes very basic information such as name, job title, company, and contact information. While using the API provides access to everything that would be visible to you as an authenticated user browsing profiles at *http://linkedin.com*, we can get all of the job title details we need for this first exercise by exporting as address book data in a comma-separated values (CSV) file format. To initiate the export, select the "Connections" menu item from the "Contacts" menu and select the "Export connections" link from within your LinkedIn account. Alternatively, you can navigate directly to the Export LinkedIn Connections page. We'll be using Python's `csv` module that's part of Python's standard library to parse the exported data, so in order to ensure compatibility with the upcoming code listing, choose the "Outlook CSV" option from the available choices. If you haven't already, you'll need to `pip install prettytable`, a package that produces nicely formatted tabular output.

Assuming you have a comma-separated values (CSV) file of contacts that you've exported from LinkedIn, you could perform some basic normalization and print out selected entities from a histogram, as illustrated in Example 3-5.

Example 3-5. Simple normalization of company suffixes from address book data

```
import csv
from collections import Counter
```

```
from operator import itemgetter
from prettytable import PrettyTable

# XXX: Set this path to wherever your "Outlook CSV"
# formatted file of connections from
# http://www.linkedin.com/people/export-settings
# is located.

CSV_FILE = ''


# Define a set of transforms that converts the first item
# to the second item. Here, we're simply handling some
# commonly known abbreviations, stripping off common suffixes,
# etc.

transforms = [(', Inc.', ''), (', Inc', ''), (', LLC', ''), (', LLP', ''),
              (' LLC', ''), (' Inc.', ''), (' Inc', '')]

csvReader = csv.DictReader(open(CSV_FILE), delimiter=',', quotechar='"')
contacts = [row for row in csvReader]
companies = [c['Company'].strip() for c in contacts if c['Company'].strip() != '']

for i in range(len(companies)):
    for transform in transforms:
        companies[i] = companies[i].replace(*transform)

pt = PrettyTable(field_names=['Company', 'Freq'])
pt.align = 'l'
c = Counter(companies)
[pt.add_row([company, freq])
 for (company, freq) in sorted(c.items(), key=itemgetter(1), reverse=True)
     if freq > 1]
print pt
```

Sample results for this script follow:

```
    +-------------------------------+------+
    | Company                       | Freq |
    +-------------------------------+------+
    | Digital Reasoning Systems     | 31   |
    | O'Reilly Media                | 19   |
    | Google                        | 18   |
    | Novetta Solutions             | 9    |
    | Mozilla Corporation           | 9    |
    | Booz Allen Hamilton           | 8    |
    |       ...                     | ...  |
    +-------------------------------+------+
```

Keep in mind that you'll need to get a little more sophisticated to handle more complex situations, such as the various manifestations of company names that have evolved over the years like O'Reilly Media that you might encounter. For example, you might see this

company's name represented as O'Reilly & Associates, O'Reilly Media, O'Reilly, Inc.,[3] or just O'Reilly.

Normalizing and Counting Job Titles

As might be expected, the very same problem in normalizing company names presents itself in considering job titles, except that it can get a lot messier because job titles are so much more variable. Table 3-1 lists a few job titles you're likely to encounter in a software company that include a certain amount of natural variation. How many distinct roles do you see for the 10 distinct titles that are listed?

Table 3-1. Example job titles for the technology industry

Job Title
Chief Executive Officer
President/CEO
President & CEO
CEO
Developer
Software Developer
Software Engineer
Chief Technical Officer
President
Senior Software Engineer

While it's certainly possible to define a list of aliases or abbreviations that equate titles like CEO and Chief Executive Officer, it may not be very practical to manually define lists that equate titles such as Software Engineer and Developer for the general case in all possible domains. However, for even the messiest of fields in a worst-case scenario, it shouldn't be too difficult to implement a solution that condenses the data to the point that it's manageable for an expert to review it, and then feed it back into a program that can apply it in much the same way that the expert would have done. More times than not, this is actually the approach that organizations prefer since it allows humans to briefly insert themselves into the loop to perform quality control.

Recall that one of the most obvious starting points when working with any data set is to count things, and this situation is no different. Let's reuse the same concepts from normalizing company names and implement a pattern for normalizing common job titles and then performs a basic frequency analysis on those titles as an initial basis for

3. If you think this is starting to sound complicated, just consider the work taken on by Dun & Bradstreet (the "Who's Who" of company information), the company that is blessed with the challenge of maintaining a worldwide directory that identifies companies spanning multiple languages from all over the globe.

clustering. Assuming you have a reasonable number of exported contacts, the minor nuances among job titles that you'll encounter may actually be surprising, but before we get into that, let's introuduce some sample code that establishes some patterns for normalizing record data and takes a basic inventory sorted by frequency. Example 3-6 inspects job titles and prints out frequency information for the titles themselves and for individual tokens that occur in them.

Example 3-6. Standardizing common job titles and computing their frequencies

```
import csv
from operator import itemgetter
from collections import Counter
from prettytable import PrettyTable

CSV_FILE = '/Users/matthew/Downloads/linkedin_connections_export_microsoft_outlook (1).csv'

transforms = [
    ('Sr.', 'Senior'),
    ('Sr', 'Senior'),
    ('Jr.', 'Junior'),
    ('Jr', 'Junior'),
    ('CEO', 'Chief Executive Officer'),
    ('COO', 'Chief Operating Officer'),
    ('CTO', 'Chief Technology Officer'),
    ('CFO', 'Chief Finance Officer'),
    ('VP', 'Vice President'),
    ]

csvReader = csv.DictReader(open(CSV_FILE), delimiter=',', quotechar='"')
contacts = [row for row in csvReader]

# Read in a list of titles and split apart
# any combined titles like "President/CEO"
# Other variations could be handled as well such
# as "President & CEO", "President and CEO", etc.

titles = []
for contact in contacts:
    titles.extend([t.strip() for t in contact['Job Title'].split('/')
                   if contact['Job Title'].strip() != ''])

# Replace common/known abbreviations

for i in range(len(titles)):
    for transform in transforms:
        titles[i] = titles[i].replace(*transform)

# Print out a table of titles sorted by frequency

pt = PrettyTable(field_names=['Title', 'Freq'])
pt.align = 'l'
c = Counter(titles)
```

```
[pt.add_row([title, freq])
 for (title, freq) in sorted(c.items(), key=itemgetter(1), reverse=True)
     if freq > 1]
print pt

# Print out a table of tokens sorted by frequency

tokens = []
for title in titles:
    tokens.extend([t.strip(',') for t in title.split()])
pt = PrettyTable(field_names=['Token', 'Freq'])
pt.align = 'l'
c = Counter(tokens)
[pt.add_row([token, freq])
 for (token, freq) in sorted(c.items(), key=itemgetter(1), reverse=True)
     if freq > 1 and len(token) > 2]
print pt
```

In short, the code reads in CSV records and makes a mild attempt at normalizing them by splitting apart combined titles that use the forward slash (like a title of "President/CEO") and replacing known abbreviations. Beyond that, it just displays the results of a frequency distribution of both full job titles and individual tokens contained in the job titles. This is nothing terribly advanced beyond the previous exercise with company names, but it serves as a useful starting template and provides you with some reasonable insight into how the data breaks down. Sample results follow:

```
+-----------------------------------+------+
| Title                             | Freq |
+-----------------------------------+------+
| Chief Executive Officer           | 19   |
| Senior Software Engineer          | 17   |
| President                         | 12   |
| Founder                           | 9    |
| ...                               | ...  |
+-----------------------------------+------+

+---------------+------+
| Token         | Freq |
+---------------+------+
| Engineer      | 43   |
| Chief         | 43   |
| Senior        | 42   |
| Officer       | 37   |
| ...           | ...  |
+---------------+------+
```

So what does the data show us? One thing that's interesting is that the sample results show that the most common job title based on exact matches is "Chief Executive Officer," which is closely followed by other senior positions such as "President" and "Founder," so the ego of this professional network has reasonably good access to enterpreneurs and

business leaders. The most common token from within the job titles is "Engineer" and "Chief." The "Chief" token correlates back to the previous thought about connections to higher-ups in companies, while the token "Engineer" provides a slightly different clue into the nature of the professional network. Although "Engineer" is not a constituent token of the most common job title, it does appear in a large number of job titles such as "Senior Software Engineer" and "Software Engineer," which show up near the top of the job titles list. Therefore, the ego of this network appears to have connections into a highly technical audience as well. In analyzing job title or address book data, this is precisely the kind of insight that motivates the need for an approximate matching or clustering algorithm. The next section investigates further.

Normalizing and Counting Locations

Although LinkedIn includes a general geographic region, usually corresponding to a metropolitan area, for each of your connections, this label is not specific enough that can trivially be pinpointed on a map without some additional work. For example, knowing the someone works in the "Greater Nashville Area" is useful and, as a human being with additional knowledge, we know that this label probably refers to Nashville, TN metro area. However, writing code to transform "Greater Nashville Area" to a set of coordinates that you could render on a map can be trickier than it sounds, especially when the human readable label for a region is especially ambiguous. For example, the population of New York City might be high enough that you can reasonably infer that "New York" refers to New York City, New York, but what about "Smithville"? There are hundreds of Smithvilles in the United States, and many states even have more than one Smithville, so it could take geographic context beyond the surrounding state to make the right determination. It may not be the case that "Greater Smithville Area" is something you'll see on LinkedIn, but it serves to illustrate the general problem of disambiguating a geographic reference so that it can be resolved to a specific set of coordinates. As a generalized problem, disambiguating geographic references is quite difficult.

Disambiguating and geocoding the whereabouts of LinkedIn connections is slightly easier than the most generalized form of the problem because most professionals tend to identify with the larger metropolitan area that they're associated with, and there are a relatively finite number of these regions. Although not always the case, you can generally employ the crude assumption that the location referred to in a LinkedIn profile is a relatively well-known location and is likely to be the "most popular" metropolitan region by that name, and oftentimes, additional context is provided for slightly lesser known locations.

A python package called `geopy` can be installed via `pip install geopy` and provides a generalized mechanism for passing in labels for locations, and getting back lists of coordinates that might match. The `geopy` package itself is a proxy to multiple web services providers such as Bing, Google, and others that perform the geocoding, and an advantage of using it is that it provides a standardized API for interfacing with various geo-

coding services so that you don't have to manually craft requests and parse responses. The geopy GitHub code repository is a good strting point for reading the documentationt that's available online.

Example 3-7 illustrates how to use `geopy` with Microsoft's Bing, which offers a generous number of API calls for accounts that fall under educational usage guidelines such as this book[4]. In order to run the script, you will need to request an API key from a service provider such as Bing

Example 3-7. Geocoding locations with Microsoft Bing

```
from geopy import geocoders

GEO_APP_KEY = '' # XXX: Get this from https://www.bingmapsportal.com
g = geocoders.Bing(GEO_APP_KEY)
print g.geocode("Nashville", exactly_one=False)
```

The keyword parameter `exactly_one=False` tells the geocoder not to trigger an error if there is more than one possible result, which will be more common than you might imagine. Sample results from this script follow and illustrate the nature of using an ambiguous label like "Nashville" to resolve a set of coordinates:

```
[(u'Nashville, TN, United States', (36.16783905029297, -86.77816009521484)),
        (u'Nashville, AR, United States', (33.94792938232422, -93.84703826904297)),
        (u'Nashville, GA, United States', (31.206039428710938, -83.25031280517578)),
        (u'Nashville, IL, United States', (38.34368133544922, -89.38263702392578)),
        (u'Nashville, NC, United States', (35.97433090209961, -77.96495056152344))]
```

The Bing geocoding service appears to return the most populous locations first in the list of results, so we'll opt to simply select the first item in the list as our response given the nature of how LinkedIn generally exposes locations in profiles as large metropolitan areas. However, before we'll be able to geocode, we'll have to return to the problem of data normalization, because passing in a value such as "Greater Nashville Area" to the geocoder won't return a response to us. (Try it and see for yourself.) As a pattern, we can transform locations such that common prefixes and suffixes are stripped as a routine matter as illustrated in Example 3-8.

Example 3-8. Geocoding locations of LinkedIn connections with Microsoft Bing

```
from geopy import geocoders

GEO_APP_KEY = '' # XXX: Get this from https://www.bingmapsportal.com
g = geocoders.Bing(GEO_APP_KEY)
```

4. Bing is the recommended geocoder for exercises in this book with `geopy`, because at the time of this writing, the Yahoo! geocoding service was not operational due to some changes in product strategy resulting in the creation of a new product called Yahoo! BOSS Geo Services. Although the Google Maps (V3) API was operational, its maximum number of requests per day seemed less ideal than the ones offered by Bing.

```
transforms = [('Greater ', ''), (' Area', '')]

results = {}
for c in connections:
    if not c.has_key('location'): continue

    transformed_location = c['location']['name']
    for transform in transforms:
        transformed_location = transformed_location.replace(*transform)

    results.update({ c['location']['name'] : g.geocode(transformed_location, exactly_one=False)[0] })

print json.dumps(results, indent=1)
```

Sample results from the geocoding exercise follow:

```
    {
     "Greater Chicago Area": [
      "Chicago, IL, United States",
      [
       41.884151458740234,
       -87.63240814208984
      ]
     ],
     "Greater Boston Area": [
      "Boston, MA, United States",
      [
       42.3586311340332,
       -71.05670166015625
      ]
     ],
     "Bengaluru Area, India": [
      "Bangalore, Karnataka, India",
      [
       12.966970443725586,
       77.5872802734375
      ]
     ],
     "San Francisco Bay Area": [
      "CA, United States",
      [
       37.71476745605469,
       -122.24223327636719
      ]
     ],
     ...
    }
```

Later in this chapter, we'll use the coordinates returned from geocoding as part of a clustering algorithm that can be an interesting way to analyze your professional network. In the meanwhile, there's another interesting visualization called a cartogram that can be interesting for visualizing your network.

Visualizing Locations with Cartograms

A cartogram is a visualization that displays a geography by scaling geographic boundaries according to an underlying variable. For example, a map of the United States might scale the size of each state so that it is larger or smaller than it should be based upon a variable such as obesity rate, poverty levels, number of millionaires, or any other variable. The resulting visualization would not necessarily present a fully integrated view of the geography since the individual states would no longer fit together due to their scaling. Still, you'd have a very intuitive idea about the overall status of the variable that led to the scaling for the state. A specialized variation of a cartogram called a Dorling Cartogram substitutes a shape such as a circle for each unit of area on a map in its approximate location and scales the size of the shape according to the unerlying variable. Another way to describe a Dorling Cartogram might be to call it a "geographically clustered bubble chart." They're a great visualization tool because they allow you to use your instincts about where information should appear on a 2D mapping surface, and they are able to encode parameters using very intuitive properties of shapes, like area and color.

Given that the results from the Bing geocoding service return results that include the state for each city that is geocoded, let's take advantage of this information and build a Dorling Cartogram of your professional network where we'll scale the size of each state according to the number of contacts you have from that state. D3, a cutting-edge visualization toolkit introduced in Chapter 2, includes most of the machinery for a Dorling Cartogram and provides a highly customizable means of extending the visualization to include other varibles if you'd like to do so. D3 also includes several other visualizations that convey geographical information, such as heatmaps, symbol maps, chloropleth maps, and many other visualizations that should be easily adapted to the working data.

There's really just one data munging nuance that needs to be performed in order to visualize your contacts by state, and that's the task of parsing the states from the geocoder responses. In general, there can be some slight variation in the text of the response that contains the state, but as a general patter, the state is always represented by two consecutive uppercase letters, and it turns out that a regular expression is a fine way to parse out that kind of pattern from text. Example 3-9 illustrates how to use the `re` package from Python's standard library to parse the geocoder response and write out a JSON file that can be loaded by a D3-powered Dorling Cartogram visualization. Teaching regular expression fundamentals is outside the current scope of our discussion, but the gist of the pattern `'.*([A-Z]{2}).*'` is that we are looking for exactly two consecutive uppercase letters in the text, which can be preceeded or followed by any text at all as denoted by the `.*` wildcard. Parentheses are used to capture the group that we are interested in so that it can easily be retrieved.

Example 3-9. Parsing out states from Bing geocoder results using a regular expression

```
import re

# Most results contain a response that can be parsed by
# picking out the first to consecutive upper case letters
# as a clue for the state
pattern = re.compile('.*([A-Z]{2}).*')

def parseStateFromBingResult(r):
    result = pattern.search(r[0][0])
    if len(result.groups()) == 1:
        return result.groups()[0]
    else:
        print "Unresolved match:", result.groups()
        return "???"


transforms = [('Greater ', ''), (' Area', '')]

results = {}
for c in connections:
    if not c.has_key('location'): continue
    if not c['location']['country']['code'] == 'us': continue

    transformed_location = c['location']['name']
    for transform in transforms:
        transformed_location = transformed_location.replace(*transform)

    results.update({c['location']['name'] :
                    parseStateFromBingResult(g.geocode(transformed_location, exactly_one=False))})

print json.dumps(results, indent=1)
```

Sample results follow and illustrate the efficacy of this technique:

```
    {
     "Greater Chicago Area": "IL",
     "Greater Boston Area": "MA",
     "Dallas/Fort Worth Area": "TX",
     "San Francisco Bay Area": "CA",
     "Washington D.C. Metro Area": "DC",
     ...
    }
```

With the ability to distill reliable state abbreviations from your LinkedIn contacts, we can now compute the frequencies of how often each state appears, which is all that is needed to drive a turn-key Dorling Cartogram visualization with D3. A sample visualization for a professional network is displayed in Figure 3-2. In many cartograms, note that Alaska and Hawaii and often displayed in the lower-left hand corner of the visualization (as is the case with many maps that display them as inlays). Despite the fact that the visualization is just a lot of carefully displayed circles on the map, it's relatively

obvious as to which circles correspond to which states. Hovering over circles produces tooltips that display the name of the state by default, and additional customization would not be very difficult to implement by observing standard D3 best practices. The process of generating the final output for consumption by D3 involves little more than generating a frequency distribution by state and serializing it out as JSON.

Some of the code for creating a Dorling Cartogram from your LinkedIn connections is omitted from this section for brevity but is included as a completely turn-key example with the IPython Notebook for this chapter.

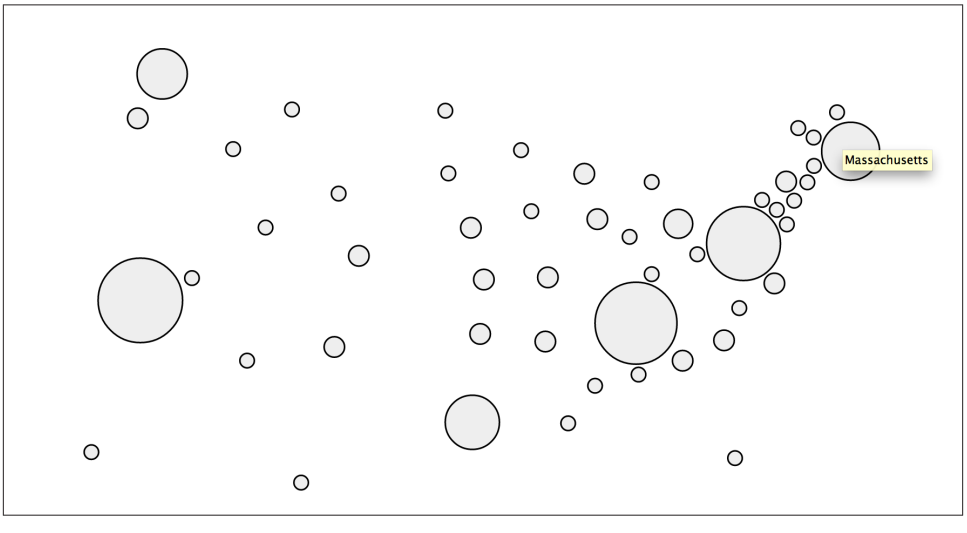


Figure 3-2. A Dorling Cartogram of locations resolved from a LinkedIn professional network. Tooltips display the name of each state when circles are hovered over.

Measuring Similarity

With an appreciation for some of the subtleties and nuances that are required to normlize data, let us now turn our attention to the problem of computing similarity, which is the principal basis of clustering. The most substantive decision that needs to be made in taking a set of strings—job titles, in this case—and clustering them in a useful way is which underlying similarity metric to use. There are myriad string similarity metrics available, and choosing the one that's most appropriate for your situation largely depends on the nature of your objective. Although these similarity measurements are not difficult to define and compute ourselves, we'll take this opportunity to introduce NLTK (Natural Language Toolkit), a Python toolkit that you'll be glad to have in your arsenal

of tools for mining the social web. Like other python packages, simply `pip install nltk` to install NLTK and be on your merry way. A few of the common similarity metrics that might be helpful in comparing job titles that are implemented in NLTK follow:

Edit distance
: Edit distance, also known as Levenshtein distance, is a simple measure of how many insertions, deletions, and replacements it would take to convert one string into another. For example, the cost of converting "dad" into "bad" would be one replacement operation (substituting the first "d" for a "b") and would yield a value of 1. NLTK provides an implementation of edit distance via the `nltk.metrics.distance.edit_distance` function. It's an important detail to note that the actual edit distance between two strings is quite different from the number of operations required to compute the edit distance; computation of edit distance is usually on the order of *M*N* operations for strings of length *M* and *N*. In other words, computing edit distance can be a computationally intense operation, so use it wisely on nontrivial amounts of data.

n-gram similarity
: An *n*-gram is just a terse way of expressing each possible consecutive sequence of *n* tokens from a text, and it provides the foundational data structure for computing collocations. There are many variations of *n*-gram similarity, but consider the straightforward case of computing all possible bigrams (2-grams) for the tokens of two strings, and scoring the similarity between the strings by counting the number of common bigrams between them. Example 3-10 demonstrates.

An extended discussion of *n*-grams and collocations is presented in "Bigram Analysis" on page 154 as part of the next chapter.

Example 3-10. Using NLTK to compute bigrams in the Python interpreter

```
>>> ceo_bigrams = nltk.bigrams("Chief Executive Officer".split(), pad_right=True, \
... pad_left=True)
>>> ceo_bigrams
[(None, 'Chief'), ('Chief', 'Executive'), ('Executive', 'Officer'),
('Officer', None)]
>>> cto_bigrams = nltk.bigrams("Chief Technology Officer".split(), pad_right=True, \
... pad_left=True)
>>> cto_bigrams
[(None, 'Chief'), ('Chief', 'Technology'), ('Technology', 'Officer'),
 ('Officer', None)]
>>> len(set(ceo_bigrams).intersection(set(cto_bigrams)))
2
```

Note that the use of the keyword arguments `pad_right` and `pad_left` intentionally allows for leading and trailing tokens to match. The effect of padding is to allow bigrams such as (None, 'Chief') to emerge, which are important matches across job titles. NLTK provides a fairly comprehensive array of bigram and trigram (3-gram) scoring functions via the `BigramAssociationMeasures` and `TrigramAssociationMeasures` classes defined in its `nltk.metrics.association` module.

Jaccard distance

More often than not, similarity can be computed between two sets of things, where a set is just an unordered collection of items. The Jaccard metric expresses the similarity of two sets and is defined by the intersection of the sets divided by the union of the sets. Mathematically, this formula is written as |Set1 intersection Set2/| Set1 union Set2|. In other words, the Jaccard distance is the number of items in common between the two sets (their intersection) divided by the total number of distinct items in the two sets (their union.) In general, you'll compute Jaccard similarity by using *n*-grams, including unigrams (single tokens), to measure the similarity of two strings. An implementation of Jaccard distance is available at `nltk.metrics.distance.jaccard_distance`, which is simply implemented as:

```
len(X.union(Y)) - len(X.intersection(Y)))/float(len(X.union(Y))
```

where `X` and `Y` are instances of Python's `set`.

We're Gonna Analyze Like It's 1874

Although the concepts involved in set theory are as old as time itself, it is Georg Cantor who is generally credited with inventing set theory. His paper, "On a Characteristic Property of All Real Algebraic Numbers," written in 1874, formalized set theory as part of his work on answering questions related to the concept of infinity. For example:

- Are there more natural numbers (zero and the positive integers) than integers (positive and negative numbers)?
- Are there more rational numbers (numbers that can be expressed as fractions) than integers?
- Are there more irrational numbers (numbers that cannot be expressed as fractions, such as pi, √2, etc.) than rational numbers?

The gist of Cantor's work around infinity as it relates to the first two questions is that the cardinalities of the sets of natural numbers, integers, and rational numbers are all equal because you can map these numbers such that they form a sequence with a definite starting point that extends forever in *one* direction. Even though there is never an ending point, the cardinalities of these sets are said to be countably infinite because there is a definite sequence that could be followed deterministically if you simply had enough time to count them. The cardinality of a countably infinite

set became known by mathematicians as $\aleph_0$, an official definition of infinity. Consider the following numeric patterns that convey the basic idea behind countably infinite sets. Each pattern shows a starting point that can extend infinitely:

- Natural numbers: 0, 1, 2, 3, 4, …
- Positive integers: 1, 2, 3, 4, 5, …
- Negative integers: -1, -2, -3, -4, -5, …
- Integers: 0, 1, -1, 2, -2, 3, -3, 4, -4, …
- Rational numbers: 0/0, 0/1, -1/1, -1/0, -1/-1, 0/-1, 1/-1, 1/0, 1/1, …

Notice that the pattern for the rational numbers is that you can start at the origin of the Cartesian plane and build out a spiral in which each *x/y* coordinate pair is expressed as a fraction, which is a rational number. (The two cases where it is undefined because of division by zero are of no consequence to the cardinality of the set as a whole.)

As it runs out, however, the cardinality of the set of irrational numbers is not equal to $\aleph_0$, because it is impossible to arrange them in such a way that they are countable and form a one-to-one correspondence back to a set having cardinality $\aleph_0$. Cantor used what became known as the famous *diagonalization argument* as the proof. The gist of the diagonalization proof is that just when you think you've mapped out a sequence that makes the irrational numbers countable, it can be shown that a whole slew of numbers are missing—and when you put these missing numbers into the sequence, there are still a slew of numbers missing. As it turns out, you can never form the one-to-one correspondence that's necessary. Thus, the cardinality of the set of irrational numbers is not the same as the cardinalities of the sets of natural numbers, positive integers, and irrational numbers, because a one-to-one correspondence from one of these sets cannot be derived.

So what is the cardinality of the set of irrational numbers? It can be shown that the power set of the set having cardinality $\aleph_0$ is the cardinality of the set of all irrational numbers. This value is known as $\aleph_1$. Further, the power set of the set having cardinality $\aleph_1$ is known as $\aleph_2$, etc. Computing the power set of a set of infinite numbers is admittedly a difficult concept to wrap one's head around, but it's one well worth pondering when you're having trouble sleeping at night.

MASI distance

The MASI[5] distance metric is a weighted version of Jaccard similarity that adjusts the score to result in a smaller distance than Jaccard when a partial overlap between

5. Measuring Agreement on Set-Valued Items. See *http://www.cs.columbia.edu/~becky/pubs/lrec06masi.pdf*.

sets exists. MASI distance is defined in NLTK as `nltk.metrics.distance.masi_distance`, which can be implemented as:

```
1 - float(len(X.intersection(Y)))/float(max(len(X),len(Y)))
```

where `X` and `Y` are Python `sets` of items. Example 3-11 and the sample results in Table 3-2 provide some data points that may help you better understand this metric.

Example 3-11. Using built-in distance metrics from NLTK to compare small sets of items

```
from nltk.metrics.distance import jaccard_distance, masi_distance
from prettytable import PrettyTable

field_names = ['X', 'Y', 'Jaccard(X,Y)', 'MASI(X,Y)']
pt = PrettyTable(field_names=field_names)
pt.align = 'l'

for z in range(4):
    X = set()
    for x in range(z, 4):
        Y = set()
        for y in range(1, 3):
            X.add(x)
            Y.add(y)
            pt.add_row([list(X), list(Y), round(jaccard_distance(X, Y), 2),
                        round(masi_distance(X, Y), 2)])
print pt
```

Notice that whenever the two sets are either completely disjoint or equal—i.e., when one set is totally subsumed[6] by the other—the MASI distance works out to be the same value as the Jaccard distance. However, when the sets overlap only partially, the MASI distance works out to be higher than the Jaccard distance. Whether you think MASI will be more effective than Jaccard distance for the task of scoring two arbitrary job titles and clustering them together if the similarity (distance) between them exceeds a certain threshold is up for discussion. There's probably not going to be a definitive answer for all situations, and it's good to have options during experimentation; hence, the introduction of this variation of Jaccard. The table of sample results in Table 3-2 tries to help you wrap your head around how the two comparisons differ.

Table 3-2. A comparison of Jaccard and MASI distances for two small sets of items as emitted from Example 3-11

X	Y	Jaccard (X,Y)	MASI (X,Y)
[0]	[1]	1.0	1.0

6. `def subsumed(X,Y): return X.union_update(Y) == X or Y.union_update(X) == Y`

X	Y	Jaccard (X,Y)	MASI (X,Y)
[0]	[1, 2]	1.0	1.0
[0, 1]	[1]	0.5	0.5
[0, 1]	[1, 2]	0.67	0.5
[0, 1, 2]	[1]	0.67	0.67
[0, 1, 2]	[1, 2]	0.33	0.33
[0, 1, 2, 3]	[1]	0.75	0.75
[0, 1, 2, 3]	[1, 2]	0.5	0.5
[1]	[1]	0.0	0.0
[1]	[1, 2]	0.5	0.5
[1, 2]	[1]	0.5	0.5
[1, 2]	[1, 2]	0.0	0.0
[1, 2, 3]	[1]	0.67	0.67
[1, 2, 3]	[1, 2]	0.33	0.33
[2]	[1]	1.0	1.0
[2]	[1, 2]	0.5	0.5
[2, 3]	[1]	1.0	1.0
[2, 3]	[1, 2]	0.67	0.5
[3]	[1]	1.0	1.0
[3]	[1, 2]	1.0	1.0

In addition to these handy similarity measurements and among its other numerous utilities, NLTK provides a class you can access as `nltk.FreqDist` that is a frequency distribution similar to the way that we've been using `collections.Counter` from Python's standard library.

While the list of interesting similarity measurements could go on and on, there's quite literally an ocean of literature about them online, and it's easy enough to plug in and try out different similarity heuristics once you have a better feel for the data you're mining. The next section works up a script that clusters job titles using these similarity measurements.

Approaches to Clustering

With the prerequisite appreciation for data normalization and similarity heuristics in place, let's now collect some real-world data from LinkedIn and compute some meaningful clusters to gain a few more insights into the dynamics of your professional network. Whether you want to take an honest look at whether your networking skills have

been helping you to meet the "right kinds of people," because you want to approach contacts who will most likely fit into a certain socioeconomic bracket with a particular kind of business enquiry or proposition, or because you want to determine if there's a better place you could live or open a remote office to drum up business development, there's bound to be something of value and interesting in a professional network laden with high quality data. The remainder of this section illustrates a few different clustering approaches by further considering the problem of grouping together job titles that are similar.

Greedy Clustering

Given MASI's partiality to partially overlapping terms, and that we have insight suggesting that overlap in titles is important, let's try to cluster job titles by comparing them to one another using MASI distance, as an extension of Example 3-6. Example 3-12 clusters similar titles and then displays your contacts accordingly. Skim the code—especially the nested loop invoking the `DISTANCE` function—and then we'll discuss.

Example 3-12. Clustering job titles using a greedy heuristic

```
import csv
from nltk.metrics.distance import masi_distance

CSV_FILE = ''

# Tweak this distance threshold and try different distance calculations
# during experimentation
DISTANCE_THRESHOLD = 0.5
DISTANCE = masi_distance

def cluster_contacts_by_title(csv_file):

    transforms = [
        ('Sr.', 'Senior'),
        ('Sr', 'Senior'),
        ('Jr.', 'Junior'),
        ('Jr', 'Junior'),
        ('CEO', 'Chief Executive Officer'),
        ('COO', 'Chief Operating Officer'),
        ('CTO', 'Chief Technology Officer'),
        ('CFO', 'Chief Finance Officer'),
        ('VP', 'Vice President'),
        ]

    seperators = ['/', 'and', '&']

    csvReader = csv.DictReader(open(csv_file), delimiter=',', quotechar='"')
    contacts = [row for row in csvReader]
```

```
# Normalize and/or replace known abbreviations
# and build up list of common titles

all_titles = []
for i in range(len(contacts)):
    if contacts[i]['Job Title'] == '':
        contacts[i]['Job Titles'] = ['']
        continue
    titles = [contacts[i]['Job Title']]
    for title in titles:
        for seperator in seperators:
            if title.find(seperator) >= 0:
                titles.remove(title)
                titles.extend([title.strip() for title in title.split(seperator)
                               if title.strip() != ''])

    for transform in transforms:
        titles = [title.replace(*transform) for title in titles]
    contacts[i]['Job Titles'] = titles
    all_titles.extend(titles)

all_titles = list(set(all_titles))

clusters = {}
for title1 in all_titles:
    clusters[title1] = []
    for title2 in all_titles:
        if title2 in clusters[title1] or clusters.has_key(title2) and title1 \
            in clusters[title2]:
            continue
        distance = DISTANCE(set(title1.split()), set(title2.split()))

        if distance < DISTANCE_THRESHOLD:
            clusters[title1].append(title2)

# Flatten out clusters

clusters = [clusters[title] for title in clusters if len(clusters[title]) > 1]

# Round up contacts who are in these clusters and group them together

clustered_contacts = {}
for cluster in clusters:
    clustered_contacts[tuple(cluster)] = []
    for contact in contacts:
        for title in contact['Job Titles']:
            if title in cluster:
                clustered_contacts[tuple(cluster)].append('%s %s'
                        % (contact['First Name'], contact['Last Name']))

return clustered_contacts
```

```
clustered_contacts = cluster_contacts_by_title(CSV_FILE)
print clustered_contacts
for titles in clustered_contacts:
    common_titles_heading = 'Common Titles: ' + ', '.join(titles)
    print common_titles_heading

    descriptive_terms = set(titles[0].split())
    for title in titles:
        descriptive_terms.intersection_update(set(title.split()))
    descriptive_terms_heading = 'Descriptive Terms: ' \
        + ', '.join(descriptive_terms)
    print descriptive_terms_heading
    print '-' * max(len(descriptive_terms_heading), len(common_titles_heading))
    print '\n'.join(clustered_contacts[titles])
    print
```

The code listing starts by separating out combined titles using a list of common conjunctions and then normalizes common titles. Then, a nested loop iterates over all of the titles and clusters them together according to a thresholded MASI similarity metric as defined by `DISTANCE`, where the assignment of `masi_distance` to `DISTANCE` was chosen to make it easy to swap in a different distance calculation for experimentation. This tight loop is where most of the real action happens in the listing: it's where each title is compared to each other title. If the distance between any two titles as determined by a similarity heuristic is "close enough," we greedily group them together. In this context, being "greedy" means that the first time we are able to determine that an item might fit in a cluster, we go ahead and assign it without further considering whether there might be a better fit, or making any attempt to account for such a better fit if one appears later. Although incredibly pragmatic, this approach produces very reasonable results. Clearly, the choice of an effective similarity heuristic is critical to its success, but given the nature of the nested loop, the fewer times we have to invoke the scoring function, the faster the code executes -- a principal concern for non-trivial sets of data. More will be said about this consideration in the next section, but do note that we do use some conditional logic to try to avoid repeating unnecessary calculations if possible.

The rest of the listing just looks up contacts with a particular job title and groups them for display, but there is one other interesting nuance involved in computing clusters: *you often need to assign each cluster a meaningful label.* The working implementation computes labels by taking the setwise intersection of terms in the job titles for each cluster, which seems reasonable given that it's the most obvious common thread. Your mileage is sure to vary with other approaches.

The types of results you might expect from this code are useful in that they group together individuals who are likely to share common responsibilities in their job duties. As previously noted, this information might be useful for a variety of reasons, whether you're planning an event that includes a "CEO Panel," trying to figure out who can best help you to make your next career move, or trying to determine whether you are really

well-enough connected to other similar professionals given your own job responsibilities and future aspirations. Abridged results for a sample professional network follow:

```
Common Titles: Chief Technology Officer,
               Founder,
               Chief Technology Officer,
               Co-Founder,
               Chief Technology Officer
Descriptive Terms: Chief, Technology, Officer
-----------------------------------------
Damien K.
Andrew O.
Matthias B.
Pete W.
...

Common Titles: Founder,
               Chief Executive Officer,
               Chief Executive Officer
Descriptive Terms: Chief, Executive, Officer
-----------------------------------------
Joseph C.
Janine T.
Kabir K.
Scott S.
Bob B.
Steve S.
John T. H.
...
```

Runtime Analysis

This section contains a relatively advanced discussion about the computational details of clustering and should be considered optional reading that may not appeal to everyone. If this is your first reading of this chapter, feel free to skip this section and peruse it upon encountering it a second time.

In the worst case, the nested loop executing the `DISTANCE` calculation from Example 3-12 would require it to be invoked in what we've already mentioned is $O(n^2)$ time complexity—in other words, `len(all_titles)*len(all_titles)` times. A nested loop that compares every single item to every single other item for clustering purposes is *not* a scalable approach for a very large value of n, but given that the unique number of titles for your professional network is not likely to be very large, it shouldn't impose a performance constraint. It may not seem like a big deal—after all, it's just a nested loop—but the crux of an $O(n^2)$ algorithm is that the number of comparisons required to process an input set increases exponentially

in proportion to the number of items in the set. For example, a small input set of 100 job titles would require only 10,000 scoring operations, while 10,000 job titles would require 100,000,000 scoring operations. The math doesn't work out so well and eventually buckles, even when you have a lot of hardware to throw at it.

Your initial reaction when faced with what seems like a predicament that scales exponentially will probably be to try to reduce the value of n as much as possible. But most of the time you won't be able to reduce it enough to make your solution scalable as the size of your input grows, because you still have an $O(n^2)$ algorithm. What you really want to do is figure out a way to come up with an algorithm that's on the order of $O(k*n)$, where k is much smaller than n and represents a manageable amount of overhead that grows much more slowly than the rate of n's growth. But as with any other engineering decision, there are performance and quality trade-offs to be made in all corners of the real world, and it sure ain't easy to strike the right balance. In fact, many data-mining companies that have successfully implemented scalable record-matching analytics at a high degree of fidelity consider their specific approaches to be proprietary information (trade secrets), since they result in definite business advantages.

For situations in which an $O(n^2)$ algorithm is simply unacceptable, one variation to the working example that you might try is rewriting the nested loops so that a random sample is selected for the scoring function, which would effectively reduce it to $O(k*n)$, if k were the sample size. As the value of the sample size approaches n, however, you'd expect the runtime to begin approaching the $O(n^2)$ runtime. Example 3-13 shows how that sampling technique might look in code; the key changes to the previous listing are highlighted in bold. The key takeaway is that for each invocation of the outer loop, we're executing the inner loop a much smaller, fixed number of times.

Example 3-13. Incorporating random sampling can improve performance of the nested loops in Example 3-12

```
# ...snip ...

all_titles = list(set(all_titles))
clusters = {}
for title1 in all_titles:
    clusters[title1] = []
    for sample in range(SAMPLE_SIZE):
        title2 = all_titles[random.randint(0, len(all_titles)-1)]
        if title2 in clusters[title1] or clusters.has_key(title2) and title1 \
            in clusters[title2]:
            continue
        distance = DISTANCE(set(title1.split()), set(title2.split()))
        if distance < DISTANCE_THRESHOLD:
            clusters[title1].append(title2)

# ...snip ...
```

Another approach you might consider is to randomly sample the data into n bins (where n is some number that's generally less than or equal to the square root of the number of items in your set), perform clustering within each of those individual bins, and then optionally merge the output. For example, if you had one million items, an $O(n^2)$ algorithm would take a trillion logical operations, whereas binning the one million items into 1,000 bins containing 1,000 items each and clustering each individual bin would require only a billion operations. (That's 1,000*1,000 comparisons for each bin for all 1,000 bins.) A billion is still a large number, but it's three orders of magnitude smaller than a trillion, and that's nothing to sneeze at.

There are many other approaches in the literature besides sampling or binning that could be far better at reducing the dimensionality of a problem. For example, you'd ideally compare every item in a set, and at the end of the day, the particular technique you'll end up using to avoid an $O(n^2)$ situation for a large value of n will vary based upon real-world constraints and insights you're likely to gain through experimentation and domain-specific knowledge. As you consider the possibilities, keep in mind that the field of machine learning offers many techniques that are designed to combat exactly these types of scale problems by using various sorts of probabilistic models and sophisticated sampling techniques. In just a moment, you'll be introduced to a fairly intuitive and well-known clustering algorithm called *k*-means, which is a general-purpose unsupervised approach for clustering a multidimensional space. We'll be using this technique later to cluster your contacts by geographic location.

Intelligent clustering enables compelling user experiences

It's easy to get so caught up in data-mining techniques such as clustering themselves that you lose sight of the business case that motivated them in the first place. Simple clustering techniques can create incredibly compelling user experiences (all over the place) by leveraging results even as simple as the job title ones we just produced. Figure 3-3 demonstrates a powerful alternative view of your data via a simple tree widget that could be used as part of a navigation pane or faceted display for filtering search criteria. Assuming that the underlying similarity metrics you've chosen have produced meaningful clusters, a simple hierarchical display that presents data in logical groups with a count of the items in each group can streamline the process of finding information, and power intuitive workflows for almost any application where a lot of skimming would otherwise be required to find the results.

The code for creating a faceted display from your LinkedIn connections is omitted from this section for brevity but is included as a completely turn-key example with the IPython Notebook for this chapter.

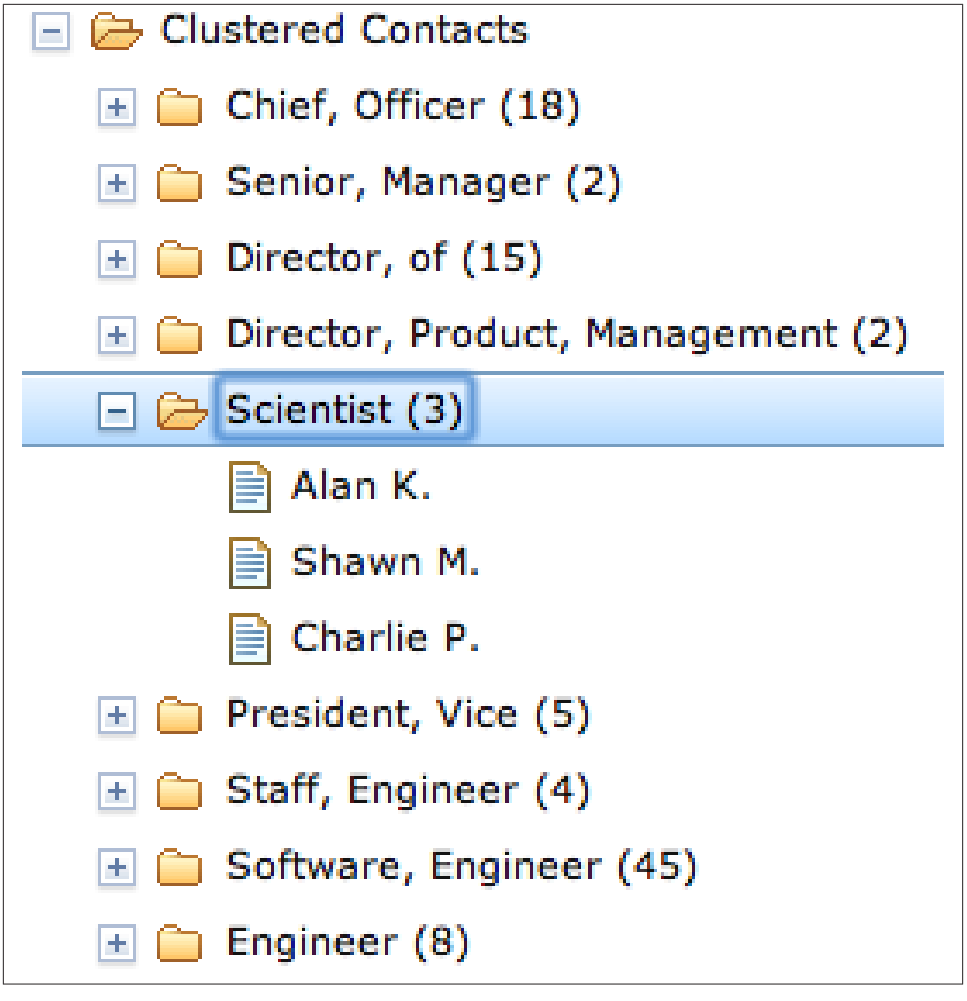


Figure 3-3. Intelligently clustering data lends itself to faceted displays and compelling user experiences

The code to create a simple navigational display can be surprisingly simple, given the maturity of Ajax toolkits and other UI libraries, and there's incredible value in being able to create user experiences that present data in intuitive ways that power workflows. Something as simple as an intelligently crafted hierarchical display can inadvertently motivate users to spend more time on a site, discover more information than they normally would, and ultimately realize more value in the services the site offers.

Hierarchical Clustering

Example 3-12 introduced an intuitive, greedy approach to clustering, principally as part of an exercise to teach you about the underlying aspects of the problem so that you have a solid grasp on what undergirds the problem. With a proper appreciation for the fundamentals now in place, it's time to introduce you to two very common clustering algorithms that you'll routinely encounter throughout your data mining career and apply in a variety of situations: hierarchical clustering and *k*-means clustering. Hierarchical clustering is superficially similar to the greedy heuristic we have been using, while *k*-means clustering is radically different. We'll primarily focus on *k*-means throughout the rest of this chapter, but it's worthwhile to briefly introduce the theory behind both of these approaches since you're very likely to encounter them during literature review and research. An excellent implementation of both of these approaches is available via the `cluster` module that you can install via `pip install cluster`.

Hierarchical clustering is a deterministic technique in that it computes the full matrix[7] of distances between all items and then walks through the matrix clustering items that meet a minimum distance threshold. It's hierarchical in that walking over the matrix and clustering items together naturally produces a tree structure that expresses the relative distances between items. In the literature, you may see this technique called agglomerative, because the leaves on the tree it constructs represent the items that are being clustered, while nodes in the tree agglomerate these items into clusters. This technique is similar to but not fundamentally the same as the approach used in Example 3-12, which uses a greedy heuristic to cluster items instead of successively building up a hierarchy. As such, the actual wall clock time (the amount of time it takes for the code to run) for hierarchical clustering may be considerably longer, and you may need to tweak your scoring function and distance threshold accordingly[8]. Oftentimes, agglomerative clustering is not appropriate for large data sets because of its impractical runtimes.

If we were to rewrite Example 3-12 to use the `cluster` package, the nested loop performing the clustering `DISTANCE` computation would be replaced with something like the code in Example 3-14.

7. The computation of a full matrix implies a polynomial runtime. For agglomerative clustering, the runtime is often on the order of $O(n^3)$.

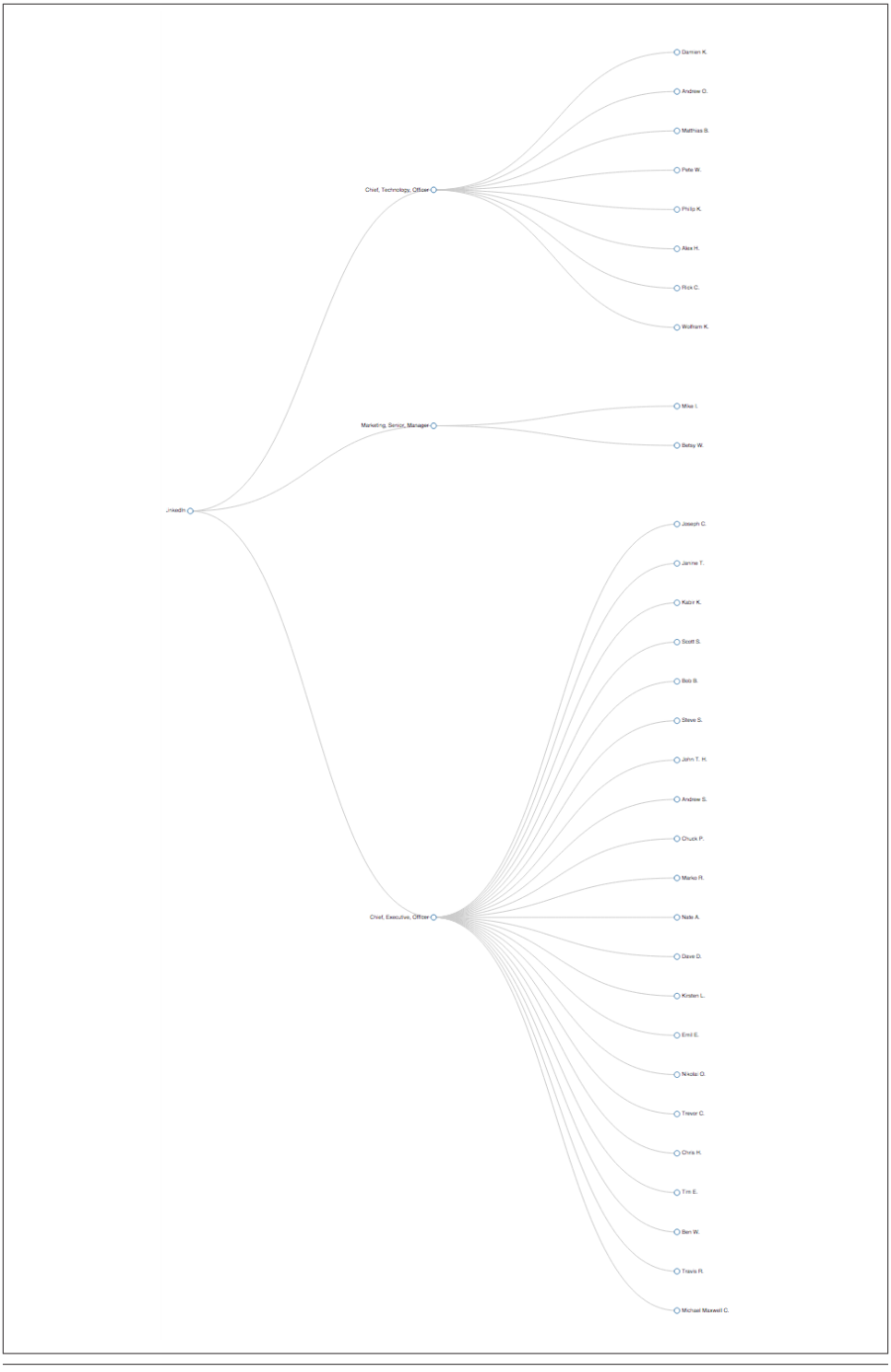
8. The use of dynamic programming and other clever bookkeeping techniques can result in substantial savings in wall clock execution time, and one of the advantages of using a well-implemented toolkit is that it is often the case that these clever optimizations are already implemented for you. For example, given that the distance between two items such as job titles is almost certainly going to be symmetric, you should only have to compute one half of the distance matrix instead of the full matrix. Therefore, even though the time complexity of the algorithm as a whole is still $O(n^2)$, only $0.5*n^2$ units of work are being performed instead of n^2 units of work.

Example 3-14. A minor modification to Example 3-12 that uses `cluster.HierarchicalClustering` instead of a greedy heuristic

```
# ... snip ...

# Define a scoring function
def score(title1, title2):
    return DISTANCE(set(title1.split()), set(title2.split()))

# Feed the class your data and the scoring function
hc = HierarchicalClustering(all_titles, score)

# Cluster the data according to a distance threshold
clusters = hc.getlevel(DISTANCE_THRESHOLD)

# Remove singleton clusters
clusters = [c for c in clusters if len(c) > 1]

# ... snip ...
```

If you're very interested in variations on hierarchical clustering, be sure to check out the `HierarchicalClustering` class's `setLinkageMethod` method, which provides some subtle variations on how the class can compute distances between clusters. For example, you can specify whether distances between clusters should be determined by calculating the shortest, longest, or average distance between any two clusters. Depending on the distribution of your data, choosing a different linkage method can potentially produce quite different results. Figure 3-4 displays a slice from a professional network as a node-link tree layout and a dendogram using D3, a state-of-the-art visualization toolkit. The node-link tree layout is more space-efficient and probably a better choice for this particular data set, while a dendogram would be a great choice if you needed to easily find correlations between each level in the tree (which would correspond to each level of agglomeration in hierarchical clustering) for a more complex set of data. If the hierarchical layout were deeper, the dendogram would have obvious benefits, but the current clustering approach is just a couple of levels deep, so the particular advantages of one layout versus the other may be mostly aesthetic for this particular data set. Keep in mind that both of the visualizations presented here are essentially just incarnations of the interactive tree widget from Figure 3-3. As these visualizations show, an amazing amount of information becomes apparent when you are able to look at a simple image of your professional network.

The code for creating a node-link tree and dendogram visualization with D3 is omitted from this section for brevity but is included as a completely turn-key example with the IPython Notebook for this chapter.


LinkedIn
Chief, Technology, Officer
Damien K.
Andrew O.
Matthias B.
Pete W.
Philip K.
Alex H.
Rick C.
Wolfram K.
Marketing, Senior, Manager
Mike I.
Betsy W.
Chief, Executive, Officer
Joseph C.
Jamie T.
Kabir K.
Scott S.
Bob B.
Steve S.
John T. H.
Andrew S.
Chuck P.
Marko R.
Nate A.
Dave D.
Kristen L.
Emil E.
Nikolai O.
Trevor C.
Chris H.
Tim E.
Ben W.
Travis R.
Michael Maxwell C.

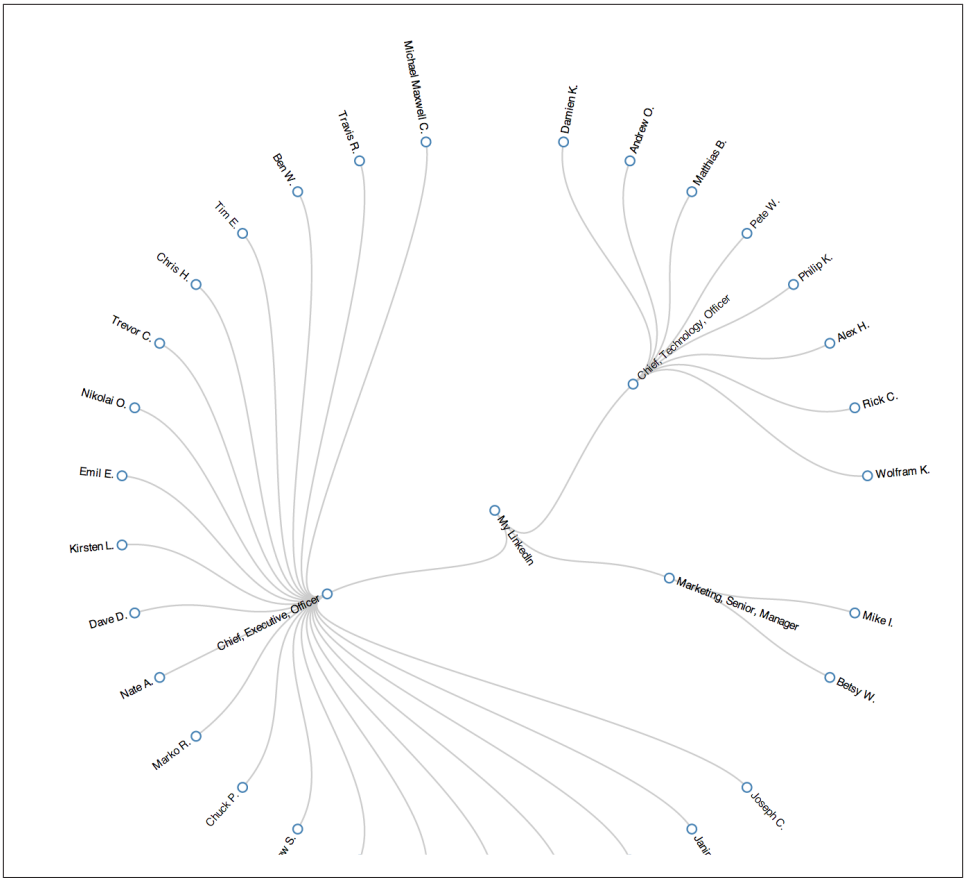


Figure 3-4. A dendogram (top) and node-link tree (bottom) layout of contacts clustered by job title

K-Means Clustering

Whereas hierarchical clustering is a deterministic technique that exhausts the possibilities and is often a very expensive computation on the order of $O(n^3)$, *k*-means clustering generally executes on the order of $O(k*n)$ times. For small values of *k*, the savings are substantial. The substantial savings in performance come at the expense of results that are approximate, but they still have the potential to be quite good. The idea is that you generally have a multidimensional space containing *n* points, which you cluster into *k* clusters through the following series of steps:

1. Randomly pick *k* points in the data space as initial values that will be used to compute the *k* clusters: K_1, K_2, …, K_k.

2. Assign each of the n points to a cluster by finding the nearest K_n—effectively creating k clusters and requiring $k*n$ comparisons.
3. For each of the k clusters, calculate the centroid, or the mean of the cluster, and reassign its K_i value to be that value. (Hence, you're computing "k-means" during each iteration of the algorithm.)
4. Repeat steps 2–3 until the members of the clusters do not change between iterations. Generally speaking, relatively few iterations are required for convergence.

Because k-means may not be all that intuitive at first glance, Figure 3-5 displays each step of the algorithm as presented in the online "Tutorial on Clustering Algorithms", which features an interactive Java applet. The sample parameters used involve 100 data points and a value of 3 for the parameter k, which means that the algorithm will produce three clusters. The important thing to note at each step is the location of the squares, and which points are included in each of those three clusters as the algorithm progresses. The algorithm only takes nine steps to complete.

Figure 3-5. Progression of k-means for k=3 with 100 points

Although you could run *k*-means on points in two dimensions or two thousand dimensions, the most common range is usually somewhere on the order of tens of dimensions, with the most common cases being two or three dimensions. When the dimensionality of the space you're working in is relatively small, *k*-means can be an effective clustering technique because it executes fairly quickly and is capable of producing very reasonable results. You do, however, need to pick an appropriate value for *k*, which is not always obvious. The remainder of this section demonstrates how to geographically cluster and visualize your professional network by applying *k*-means and rendering the output with Google Maps or Google Earth.

Clustering Your Professional Network with K-Means and Visualizing it with Google Earth

An interesting exercise in seeing *k*-means in action is to use it to visualize and cluster your professional LinkedIn network by plotting it two-dimensional space. In addition to the insight gained by visualizing how your contacts are spread out and any patterns or anomalies that intuition provides, you can analyze clusters by using your contacts, the distinct employers of your contacts, or the distinct metro areas in which your contacts reside as a basis. All three approaches might yield results that are useful for different purposes. Recalling that through the LinkedIn API you can fetch location information that describes the major metropolitan area, such as "Greater Nashville Area," we'll be able to geocode the locations into coordinates and emit them in an appropriate format such as KML that we can plot in a tool like Google Earth, which provides a very interactive user experience.

Google Maps also provides a means of uploading a custom KML file for visualization purposes. When you are logged into your Google account, navigate to "My Places" and choose the "Create Map" option to try it out. However, be advised that it does not appear to handle the case of rendering multiple placemarkers with the same coordinates in as user friendly a way as Google Earth, which makes it rather trivial to splay out placemarkers for easier inspection.

The primary things that must be done in order to convert your LinkedIn contacts to a format such as KML include parsing out the geographic location from each of your connections' profiles and constructing the KML for a visualization such as Google Earth. Recalling that Example 3-8 demonstrated how to geocode profile information and provides a working foundation for gathering the data we'll need, and the `KMeansClustering` class of the `cluster` package can calculate clusters for us, all that's really left is to munge the data and clustering results into KML, which is a relatively uninteresting exercise with XML tools.

As in Example 3-12, most of the work involved in getting to the point where the results can be visualized is data-processing boilerplate. The most interesting details are tucked away inside of `KMeansClustering`'s `getclusters` method call, toward the end of Example 3-15, which illustrates k-Means clustering. The approach demonstrated groups your contacts by location, clusters them, and then uses the results of the clustering algorithm to compute the centroids. Figure 3-6 illustrates sample results from running the code in (to come).

The `linkedin__kml_utility` that provides the `createKML` function is uninteresting and omitted for brevity but included with the IPython Notebook for this chapter as a turn-key example.

Example 3-15. Clustering your LinkedIn professional network based upon the locations of your connections and emitting KML output for visualization with Google Earth

```
import os
import json
from urllib2 import HTTPError
from geopy import geocoders
from cluster import KMeansClustering, centroid

# An uninteresting helper function to munge data and build up an XML tree

from linkedin__kml_utility import createKML

# XXX: Try different values for K to see the difference in clusters that emerge

K = 3

# XXX: Get an API Key and pass it in here. See https://www.bingmapsportal.com
GEO_API_KEY = ''
g = geocoders.Bing(GEO_API_KEY)

# Load this data from where you've previously stored it

CONNECTIONS_DATA = 'linkedin_connections.json'

OUT = "linkedin_clusters_kmeans.kml"

# Open up your saved connections with extended profile information
# or fetch them again from LinkedIn if you prefer

connections = json.loads(open(CONNECTIONS_DATA).read())['values']

locations = [c['location']['name'] for c in connections if c.has_key('location')]

# Some basic transforms may be necessary for geocoding services to function properly
# Here are a couple that seem to help.
```

```
transforms = [('Greater ', ''), (' Area', '')]

# Step 1 - Tally the frequency of each location

coords_freqs = {}
for location in locations:

    if not c.has_key('location'): continue

    # Avoid unnecessary I/O and geo requests by building up a cache

    if coords_freqs.has_key(location):
        coords_freqs[location][1] += 1
        continue
    transformed_location = location

    for transform in transforms:
        transformed_location = transformed_location.replace(*transform)

        # Handle potential I/O errors with a retry pattern...

        while True:
            num_errors = 0
            try:
                results = g.geocode(transformed_location, exactly_one=False)
                break
            except HTTPError, e:
                num_errors += 1
                if num_errors >= 3:
                    sys.exit()
                print >> sys.stderr, e
                print >> sys.stderr, 'Encountered an urllib2 error. Trying again...'

        for result in results:
            # Each result is of the form ("Description", (X,Y))
            coords_freqs[location] = [result[1], 1]
            break # Disambiguation strategy is "pick first"

# Step 2 - Build up data structure for converting locations to KML

# Here, you could optionally segment locations by continent
# country so as to avoid potentially finding a mean in the middle of the ocean
# The k-means algorithm will expect distinct points for each contact so build out
# an expanded list to pass it

expanded_coords = []
for label in coords_freqs:
    ((lat, lon), f) = coords_freqs[label]
    expanded_coords.append((label, [(lon, lat)] * f))  # Flip lat/lon for Google Earth

# No need to clutter the map with unnecessary placemarks...
```

```
kml_items = [{'label': label, 'coords': '%s,%s' % coords[0]} for (label,
             coords) in expanded_coords]

# It could also be interesting to include names of your contacts on the map for display

for item in kml_items:
    item['contacts'] = '\n'.join(['%s %s.' % (c['firstName'], c['lastName'])
                                  for c in connections if c.has_key('location') and c['location']['name

# Step 3 - Cluster locations and extend the KML data structure with centroids

cl = KMeansClustering([coords for (label, coords_list) in expanded_coords
                       for coords in coords_list])

centroids = [{'label': 'CENTROID', 'coords': '%s,%s' % centroid(c)} for c in
             cl.getclusters(K)]

kml_items.extend(centroids)

# Step 4 - Create the final KML output and write it to a file

kml = createKML(kml_items)

f = open(OUT, 'w')
f.write(kml)
f.close()

print 'Data written to ' + OUT
```

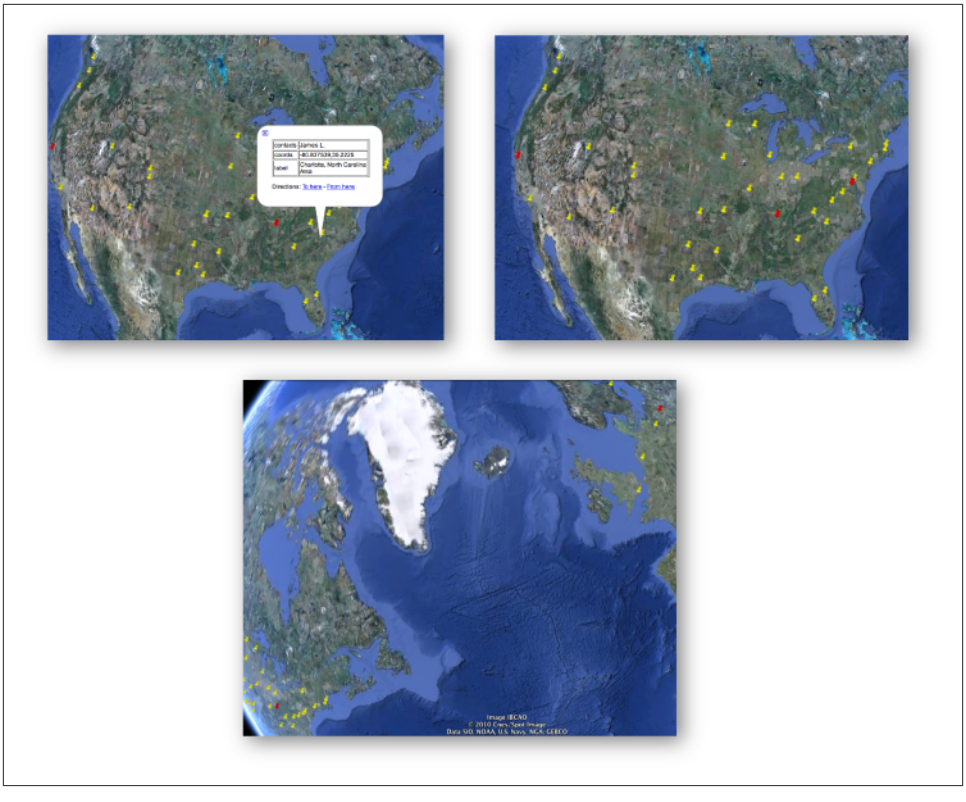


Figure 3-6. From top left to bottom: 1) clustering contacts by location so that you can easily see who lives/works in what city, 2) finding the centroids of three clusters computed by k-means, 3) don't forget that clusters could span countries or even continents when trying to find an ideal meeting location!

Just visualizing your network can be pretty interesting, but computing the geographic centroids of your professional network can also open up some intriguing possibilities. For example, you might want to compute candidate locations for a series of regional workshops or conferences. Alternatively, if you're in the consulting business and have a hectic travel schedule, you might want to plot out some good locations for renting a little home away from home. Or maybe you want to map out professionals in your network according to their job duties, or the socioeconomic bracket they're likely to fit in based on their job titles and experience. Beyond the numerous options opened up by visualizing your professional network's location data, geographic clustering lends itself to many other possibilities, such as supply chain management and Travelling Salesman types of problems.

Closing Remarks

This chapter covered some serious ground, introducing the fundamental concept of clustering and demonstrating a variety of ways to apply it to your professional network data on LinkedIn. This chapter was without a doubt more advanced than the chapters before it in terms of core content in that it began to address common problems such as data normalization for (slightly) messy data, similarity computation on normalized data, and concerns related to the computational efficiency of various approaches for a common data mining technique. Although it might be difficult to absorb all of the material on a single reading, don't be discouraged if you feel a bit overwhelemed. It may take a couple of readings to more fully absorb some of the myriad details introduced in this chapter. Also keep in mind, that a working knowledge of how to employ clustering doesn't necessarily require an advanced understanding of the theory behind it, although you should strive to understand as many of the fundamentals that undergird the techniques you employ as part of your purusits in mining the social web. As in the other chapters, you could easily make the case that we just barely touched the tip of the iceberg; there are many other interesting things that you can do with your LinkedIn data that were not introduced in this chapter, many of which involve basic frequency analyses and do not require clustering at all. That said, you do have a pretty nice powertool in your belt now.

Recommended Exercises

- Take some time to explore the extended profile information that you have available. It could be interesting to try to correlate where people work versus where they went to school and/or analyze whether people tend to relocate into and out of certain areas.
- Try employing an alterantive visualization from D3 such as a choropleth map to visualize your professional network.
- Take a look at geodict and some of the other geo utilities in the Data Science Toolkit. Can you extract locations from arbitrary prose and visualize them in a meaningful way to gain insight into what's happening in the data without having to read through all of it?
- Mine Twitter or Facebook profiles for geo information and visualize it in a meaningful way. Tweets and Facebook posts often contain geocodes as part of their structured metadata.
- The LinkedIn API provides a means of retrieving a connection's Twitter handle. How many of your LinkedIn connections have Twitter accounts associated with their professional profiles? How active are their accounts? How professional are their online Twitter personalities from the perspective of a potential employer?

- Apply clustering techniques from this chapter to tweets. Given a user's tweets, can you extract meaningful tweet entities, define a meaningful similarity computation, and cluster tweets in a meaningful way?
- Apply clustering techniques from this chapter to Facebook data such as likes or posts. Given a collections of Facebook likes for a friend, can you define a meaningful similarity computation, and cluster the likes in a meaningful way? Given all of the likes for all of your friends, can you cluster the likes (or your friends) in a meaningful way?

CHAPTER 4

Mining Google+ Activities: Computing Document Similarity, Extracting Collocations, and More

This chapter continues to build upon concepts from previous chapters and officially introduces some fundamental concepts from text mining,[1] and it's something of an inflection point in this book. Whereas we started the book with basic frequency analyses of Twitter data and gradually worked up to more sophisticated analyses of messier data from LinkedIn profiles, this chapter begins munging and making sense of textual information in documents by introducing Information Retrieval (IR) theory fundamentals such as TF-IDF, cosine similarity, and collocation detection. As you may have already inferred from the chapter title, Google+ initially serves as our primary source of data because it's inherently social, features content that's often expressed as longer form notes that resemble blog entries, and is now an established staple in the social web. It's not hard to make a case that Google+ *activities* (notes) fill an interesting niche somewhere between Twitter and blogs, and the concepts from previous chapters could equally be applied to the data behind Google+'s diverse and powerful API, although we'll opt to leave regurgitations of those exercises (mostly) as exercises that you can enjoy as recommended exercises.

Wherever possible we won't reinvent the wheel and implement analysis tools from scratch, but we will take a couple of "deep dives" when particularly foundational topics come up that are essential to an understanding of text mining. The Natural Language Toolkit (NLTK) is a powerful technology that you may recall from Chapter 3 and provides many of the tools we'll use in this chapter. Its rich suites of APIs can be a bit

1. This book avoids splitting hairs over exactly what differences could be implied by common phrases such as "text mining," "unstructured data analytics" (UDA), or "information retrieval," and simply treats them as essentially the same thing.

overwhelming at first, but don't worry: text analytics is an incredibly diverse and complex field of study, but there are lots of powerful fundamentals that can take you a long way without too significant of an investment. This chapter and the chapters after it aim to hone in on those fundamentals.

A full-blown introduction to NLTK is outside the scope of this book, but you can review the full text of Natural Language Processing with Python: Analyzing Text with the Natural Language Toolkit (O'Reilly) online.

The Google+ Platform

Anyone with a Gmail account can trivially create a Google+ account and start collaborating with friends. From a product standpoint, Google+ has evolved rapidly and used some of the most compelling features (and anti-features) of existing social network platforms such as Twitter and Facebook in carving out its own set of unique features. Like the other social web properties featured in this book, a full overview of Google+ isn't in scope for this chapter, but you can easily read about it (or sign up) online. The best way to learn is by creating an account and spending some time exploring its features. For the most part, where there's a feature, there's an API that provides that feature that you can tap into. Suffice it to say that Google+ has leveraged tried and true features of existing social networks such as marking content with hashtags and maintaining a profile according to customizable privacy settings with additional novelties such as a fresh take on content sharing called "circles," video chats called "hangouts," and extensive integration with other Google services such as Gmail contacts and Picasa web albums.

In Google+ API parlance, social interactions are framed in terms of people, activities, comments, and moments. An activity is a essentially just a note that could be just as long as a blog post, but could also be void of any real textual meaning if it were just a pointer to multimedia data such as photos or videos.

The API documentation that's available online is always the definitive source of guidance, but a brief overview may be helpful to get you thinking about how Google+ compares to another platform such as Twitter or Facebook.

People

People are Google+ users. Programatically, you'll either discover users by using the search API, look them up by a vantity URL if they're a celebrity type, or strip their Google+ ID out of the URL that appears in your web browser and use it for exploring their profile. Later in this chapter, we'll use the search API to find users on Google+.

Activities
: Activities are the things that people do on Google+. In general, people post notes that could be as long or short as they'd like (sometimes filling a niche spot between Twitter and blogs), share links or multimedia content, and so forth. Given a Google + user, you can easily retrieve a list of their activities, and we'll do this later in this chapter. Activities contain a lot of interesting metadata similar to tweets such as the number of times the activity has been reshared, etc.

Comments
: Leaving comments are the way Google+ users interact with one another. Simple statistical analysis of comments on Google+ could be very interesting and reveal a lot of insights into a person's social circles or the virality of content. For example, which other Google+ users most frequently comment on activities? Which activities have the highest numbers of comments (and why?)

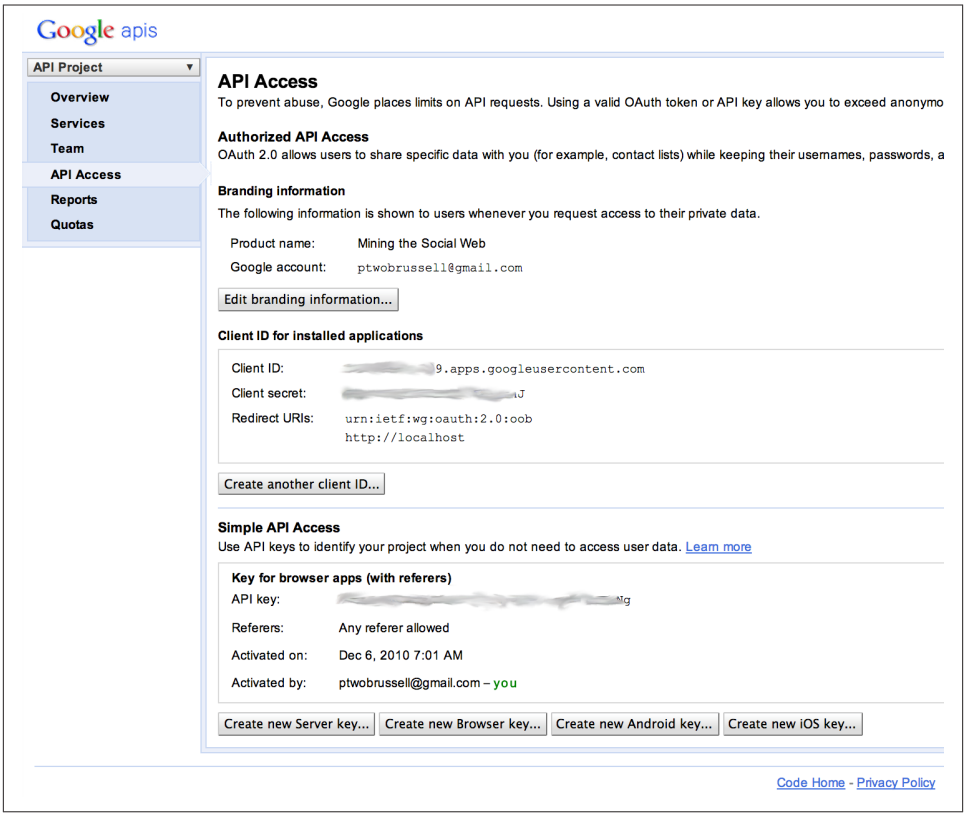
Moments
: Moments are a relatively recent addition to Google+ and represent a way of capturing interaction between a user and an Google+ application. Moments are along the same idea of the Facebook's social graph stories in that they are designed to capture and create opportunities for user interaction with the application.

For the purposes of this chapter, we'll be focusing on harvesting and analyzing Google + activity data that is textual and intended to convey the same kind of meaning that you might encounter in a tweet, blog post, or Facebook status update. In other words, we'll be trying to analyze the human language data that we can retrieve. If you haven't signed up for Google+ yet, it's worth taking the time to do so, as well as spending a few moments to familiarize yourself with a Google+ profile. One of the easiest ways to find someone on Google+ is to just search for them at *http://plus.google.com/*, and explore the platform from there.

As you go about exploring Google+, it may be helpful to bear in mind that it has a unique set of capabilities and is a little awkward to compare directly to other social web properties, so if you're expecting it to be a straightforward comparions, you might find yourself a bit surprised. It's similar to Twitter in that it provides a "following" model where you can add someone to one of your Google+ circles and keep up with their happening without any approval on their part, but the Google+ platform also offers rich integration with other Google web properties, features a powerful feature for video conferencing called hangouts, and has an API that is not all that different from Facebook in the way that users share content and interact. Without further ado, let's get busy exploring the API and work toward mining some data.

Exploring the Google+ API

From a software development standpoint, Google+ leverages OAuth like the rest of the social web to enable an application that you'll build to access data on behalf of a user, so you'll need to register an application to get appropriate credentials for accessing the Google+ Platform. The Google API Console provides a means of registering an application (called a "project" in the Google API Console) and includes the standard OAuth credentils but also exposes an API key that you can use for "simple API access", and this API key is what we'll use in this chapter to programatically access the Google+ platform and just about every other Google service. Once you've created a application, you'll also need to specifically enable it to use Google+ as a separate step. Figure 4-1 provides a screenshot of the Google+ API Console as well as a view that demonstrates what it looks like when you have enabled your application for Google+ API access.



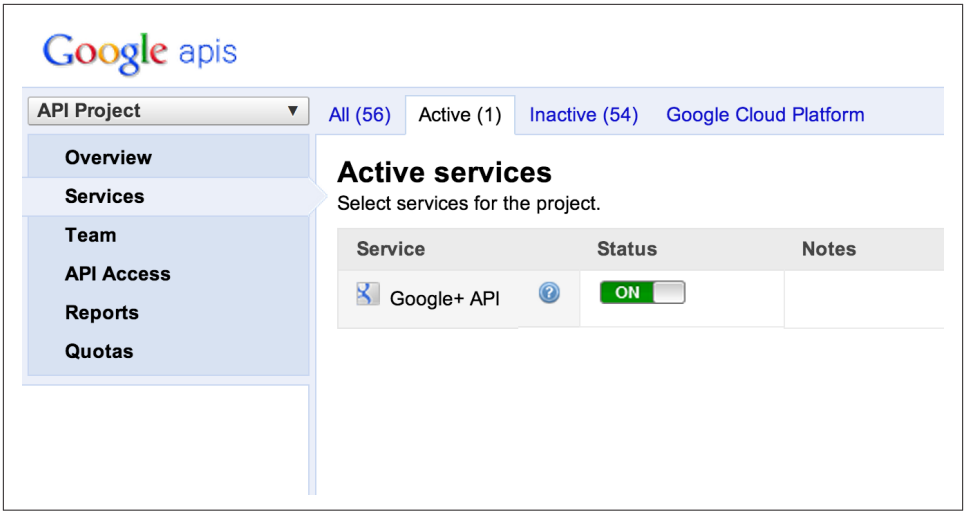


Figure 4-1. Register an application with the Google API Console to gain API access to Google services. Don't forget to enable Google+ API access as one of the services options.

A Python package called `google-api-python-client` for accessing Google's API can be installed via `pip install google-api-python-client` and is one of the standard Python-based options for accessing Google+ data. The online documentation for google-api-python-client is marginally helpful in familiarizing yourself with the capabilities of what it offers, but in general, you'll just be plugging in parameters from the official Google+ API documents into some basic access patterns with the Python package. Once you've walked through a couple of exercises, it's a relatively straightforward process since the API is predictable.

Dont' forget that `pydoc` can be helpful for gathering clues about a package, class, or method in a terminal as you are learning it. The `help` function in a standard Python interpreter is also very useful. Recall that appending `?` to a method name in IPython is a shortcut for displaying its docstring.

As an initial exercise, let's consider the problem of locating a person on Google+. Like any other social web API, Google+ offers a means of searching, and in particular, we'll be interested in the People: search API. Example 4-1 illustrates how to search for a person with the Google+ API. Since Tim O'Reilly is a well known personality with an active and interesting Google+ account, let's look him up. The basic pattern that you'll repeatedly use with the Python client is to create an instance of a service that's parameterized for Google+ access with your API key that you can then instantiate for particular platform services. Here, we create a connection to the People API by invoking `ser`

`vice.people()` and then chaining on some additional API operations that were deduced from reviewing the API documentation online. In a moment, we'll query for activity data, and you'll see that the same basic pattern holds.

Example 4-1. Searching for a person with the Google+ API

```
import httplib2
import json
import apiclient.discovery

# XXX: Enter any person's name
Q = "Tim O'Reilly"

# XXX: Enter in your API key from  https://code.google.com/apis/console
API_KEY = ''

service = apiclient.discovery.build('plus', 'v1', http=httplib2.Http(),
                                    developerKey=API_KEY)

people_feed = service.people().search(query=Q).execute()

print json.dumps(results['items'], indent=1)
```

Sample results for searching for Tim O'Reilly follow:

```
    [
     {
      "kind": "plus#person",
      "displayName": "Tim O'Reilly",
      "url": "https://plus.google.com/+TimOReilly",
      "image": {
       "url": "https://lh4.googleusercontent.com/-J8..."
      },
      "etag": "\"WIBkkymG3C8dXBjiaEVMpCLNTTs/wwgOCMn...\"",
      "id": "107033731246200681024",
      "objectType": "person"
     },
     {
      "kind": "plus#person",
      "displayName": "Tim O'Reilly",
      "url": "https://plus.google.com/11566571170551...",
      "image": {
       "url": "https://lh3.googleusercontent.com/-yka..."
      },
      "etag": "\"WIBkkymG3C8dXBjiaEVMpCLNTTs/0z-EwRK7...\"",
      "id": "115665711705516993369",
      "objectType": "person"
     },
     ...
    ]
```

The results do indeed return a list of people who all have the name Tim O'Reilly, but how would we know which one of these results refers to the well-known Tim O'Reilly of O'Reilly Media and technology fame that we are looking for? One option would be to request profile or activity information for each of these results and try to disambiguate them automatically. Another option is to render the avatars included in each of the results, which is trivial to do using a lesser-known feature of IPython Notebook. Example 4-2 illustrates how to display avatars and the corresponding id values for each search result by generating HTML and rendering it inline as a result in the notebook.

Example 4-2. Displaying avatatars for search results in IPython Notebook

```
from IPython.core.display import HTML

html = []

for p in activity_feed['items']:
    html += ['<p><img src="%s" />%s</p>' % (p['image']['url'], p['id'])]

HTML(''.join(html))
```

Sample results are displayed in Figure 4-2 and indeed provide the "quick fix" that we're looking for in our pursuit to find the particular Tim O'Reilly of O'Reilly Media.

Figure 4-2. Displaying Google+ avatars in IPython Notebook provides a quick way to disambiguate the search results and discover the person you are looking for

Although there's a multiplicity of interesting things we could do with the People API, our attention in this chapter is focusing toward an analysis of the textual content in accounts, so let's instead turn our attention to the task of retrieving activities associated with this account. As you're about to find out, Google+ Activities are the linchpin of Google+ content, containing a variety of rich content associated with the account and providing logical pivots to other platform objects such as comments. In order to get some activities, we'll need to tweak the design pattern we applied for searching for people as illustrated in Example 4-3.

Example 4-3. Fetching recent activities for a particular Google+ user

```
import httplib2
import json
import apiclient.discovery

USER_ID = '107033731246200681024' # Tim O'Reilly

# XXX: Enter in your API key from  https://code.google.com/apis/console
API_KEY = ''

service = apiclient.discovery.build('plus', 'v1', http=httplib2.Http(),
                                    developerKey=API_KEY)

activity_feed = service.activities().list(
  userId=USER_ID,
  collection='public',
  maxResults='100' # Max allowed per API
).execute()

print json.dumps(activity_feed['items'], indent=1)
```

Sample results for the first item in the results (`activity_feed['items'][0]`) follow and illustrate the basic nature of a Google+ activity:

```
    {
     "kind": "plus#activity",
     "provider": {
      "title": "Google+"
     },
     "title": "This is the best piece about privacy that I've read in a ...",
     "url": "https://plus.google.com/107033731246200681024/posts/78UeZ1jdRsQ",
     "object": {
      "resharers": {
       "totalItems": 191,
       "selfLink": "https://www.googleapis.com/plus/v1/activities/z125xvy..."
      },
      "attachments": [
       {
        "content": "Many governments (including our own, here in the US) ...",
        "url": "http://www.zdziarski.com/blog/?p=2155",
        "displayName": "On Expectation of Privacy | Jonathan Zdziarski's Domain",
```

```
    "objectType": "article"
   }
  ],
  "url": "https://plus.google.com/107033731246200681024/posts/78UeZ1jdRsQ",
  "content": "This is the best piece about privacy that I&#39;ve read ...",
  "plusoners": {
   "totalItems": 356,
   "selfLink": "https://www.googleapis.com/plus/v1/activities/z125xvyid..."
  },
  "replies": {
   "totalItems": 48,
   "selfLink": "https://www.googleapis.com/plus/v1/activities/z125xvyid..."
  },
  "objectType": "note"
 },
 "updated": "2013-04-25T14:46:16.908Z",
 "actor": {
  "url": "https://plus.google.com/107033731246200681024",
  "image": {
   "url": "https://lh4.googleusercontent.com/-J8nmMwIhpiA/AAAAAAAAAAI/A..."
  },
  "displayName": "Tim O'Reilly",
  "id": "107033731246200681024"
 },
 "access": {
  "items": [
   {
    "type": "public"
   }
  ],
  "kind": "plus#acl",
  "description": "Public"
 },
 "verb": "post",
 "etag": "\"WIBkkymG3C8dXBjiaEVMpCLNTTs/d-ppAzuVZpXrW_YeLXc5ctstsCM\"",
 "published": "2013-04-25T14:46:16.908Z",
 "id": "z125xvyidpqjdtol423gcxizetybvpydh"
}
```


At the risk of overstating the importance of really understanding the platform objects, spend some time and get to know the online documentation for Google+, especially the documentation on activities, so that you can fully appreciate the possiblities. For example, did you know that each activity object follows a 3-tuple pattern of the form (actor, verb, object)? In this post, the tuple is (Tim O'Reilly, post, note), which tells us that this paticular item in the results is a note, which is essentially just a status update with some textual content. A closer look at the result reveals that the content is something that Tim O'Reilly feels very strongly about as indicated by the title "This is the best piece abou privacy that I've read in a long time!" and clues that the note is active as evidenced by the number of reshares and comments.

A lot of interesting material appears in comments for creative content such as notes on Google+. Can you update the code to fetch the comment feed for the activity?

The astute readerm might have suspected that the `content` field for the activity contains HTML markup as evidenced by the HTML entity `I&#39;ve` that appears. In general, you should assume that the textual data exposed as Google+ activities contains some very basic markup such as `<br />` tags, escaped HTML entities for apostrophes, etc., so a little bit of additional filtering is needed to clean it up as part of best practices for data cleansing for Google+ content. Example 4-4 provides an example of how to distill plain text from the content field of a note by introducing a function called `cleanHtml` that takes advantage of a `clean_html` function provided by NLTK and another handy package for manipulating HTML called `BeautifulSoup` that converts HTML entities back to plain text. If you haven't already encountered `BeautifulSoup`, it's a package that you won't want to live without once you've added it to your toolbox due to its ability to process HTML in a reasonable way even if it is invalid and violates standards or other reasonable expectations (ala web data.) You should install these packages via `pip install nltk beautifulsoup4` if you haven't already.

Example 4-4. Cleaning HTML content in Google+ content by stripping out HTML tags and converting HTML entities back to plain text representations

```
from nltk import clean_html
from BeautifulSoup import BeautifulStoneSoup

# clean_html removes tags and
# BeautifulStoneSoup converts HTML entities

def cleanHtml(html):
  if html == "": return ""

  return BeautifulStoneSoup(clean_html(html),
          convertEntities=BeautifulStoneSoup.HTML_ENTITIES).contents[0]

print cleanHtml(activity_feed['items'][0]['object']['content'])
```

The output from the note's content once cleansed with `cleanHtml` is very clean text that can be processed without additional concerns about noise in the content. As we'll learn in this chapter and follow on chapters about text-mining, reduction of noise in text content is a critical aspect of improving accuracy. For the interested reader, the before and after content follows.

Raw content in `activity_feed['items'][0]['object']['content']`:

```
    This is the best piece about privacy that I&#39;ve read in a long time!
    If it doesn&#39;t change how you think about the privacy issue, I&#39;ll be
    surprised.  It opens:<br /><br />&quot;Many governments (including our own,
```

```
here in the US) would have its citizens believe that privacy is a switch (that
is, you either reasonably expect it, or you don't). This has been demonstrated
in many legal tests, and abused in many circumstances ranging from spying on
electronic mail, to drones in our airspace monitoring the movements of private
citizens. But privacy doesn't work like a switch - at least it shouldn't for a
country that recognizes that privacy is an inherent right. In fact, privacy,
like other components to security, works in layers...&amp;quot;<br /><br />Please read!
```

Content rendered after cleansing with `cleanHtml(activity_feed['items'][0]['object']['content'])`:

```
This is the best piece about privacy that I've read in a long time!  If it
doesn't change how you think about the privacy issue, I'll be surprised.  It opens:
"Many governments (including our own, here in the US) would have its citizens
believe that privacy is a switch (that is, you either reasonably expect it, or you
don't). This has been demonstrated in many legal tests, and abused in many
circumstances ranging from spying on electronic mail, to drones in our airspace
monitoring the movements of private citizens. But privacy doesn't work like a switch
- at least it shouldn't for a country that recognizes that privacy is an inherent right.
In fact, privacy, like other components to security, works in layers..." Please read!
```

The ability to query out clean text from Google+ is the basis for the remainder of the text mining exercises in this chapter, but on additional consideration that you may find useful before focusing our attention elsewhere is a pattern for fetching multiple pages of content. Whereas the previous example fetched 100 activities, the maximum number of results for a query, it may be the case that you'll want to iterate over an activities feed and retrieve more than the maximum number of activities per page. The pattern for pagination is outlined in the HTTP API Overview, and the Python client wrapper takes care of most of the trouble.

Example 4-5 shows how to fetch multiple pages of activities and distill the text from them if they are notes and have meaningful content.

Example 4-5. Looping over multiple pages of Google+ Activities and distilling clean text from notes

```
import httplib2
import json
import apiclient.discovery
from BeautifulSoup import BeautifulStoneSoup
from nltk import clean_html

USER_ID = '107033731246200681024' # Tim O'Reilly

# XXX: Enter in your API key from  https://code.google.com/apis/console
API_KEY = ''

MAX_RESULTS = 200 # Will require multiple requests

def cleanHtml(html):
  if html == "": return ""
```

```
  return BeautifulStoneSoup(clean_html(html),
          convertEntities=BeautifulStoneSoup.HTML_ENTITIES).contents[0]

service = apiclient.discovery.build('plus', 'v1', http=httplib2.Http(),
                                    developerKey=API_KEY)

activity_feed = service.activities().list(
  userId=USER_ID,
  collection='public',
  maxResults='100' # Max allowed per request
)

activity_results = []

while activity_feed != None and len(activity_results) < MAX_RESULTS:

  activities = activity_feed.execute()

  if 'items' in activities:

    for activity in activities['items']:

      if activity['object']['objectType'] == 'note' and \
         activity['object']['content'] != '':

        activity['title'] = cleanHtml(activity['title'])
        activity['object']['content'] = cleanHtml(activity['object']['content'])
        activity_results += [activity]

  # list_next requires the previous request and response objects
  activity_feed = service.activities().list_next(activity_feed, activities)

# Write the output to a file for convenience

f = open(USER_ID + '.json', 'w')
f.write(json.dumps(activity_results, indent=1))
f.close()

print str(len(activity_results)), "activities written to", f.name
```

With the know-how to explore the Google+ API and fetch some interesting human language data from activities' content, let's now turn our attention to the problem of analyzing the content.

Exploring Human Language Data with the Natural Language Toolkit

NLTK is written such that you can explore data very easily and begin to form some impressions without a lot of upfront investment. Before skipping ahead, though,

consider following along with the interpreter session in Example 4-6 to get a feel for some of the powerful functionality that NLTK provides right out of the box. Since you may not have done much work with NLTK before, don't forget that you can use the built-in `help` function to get more information whenever you need it. For example, `help(nltk)` would provide documentation on the NLTK package in an interpreter session.

It also might be worthwhile pointing out that not all of the functionality from NLTK is intended for incorporation into production software, since output is written through standard output and not capturable into a data structure such as a list. In that regard, methods such as `nltk.text.concordance` are considered "demo functionality." Speaking of which, many of NLTK's modules have a `demo` function that you can call to get some idea of how to use the functionality they provide, and the source code for these demos is a great starting point for learning how to use new APIs. For example, you could run `nltk.text.demo()` in the interpreter to get some additional insight into capabilities provided by the `nltk.text` module.

Example 4-6 demonstrates some good starting point for exploring the data with sample output included as part of an interactive interpreter session, and the same commands to explore the data are included in the IPython Notebook for this chapter. Are you able to follow along and understand the investigative flow of the interpreter session?

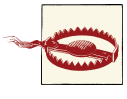
The examples throughout this chapter, including the following interpreter session, use the `split` method to tokenize text. Tokenization isn't quite as simple as splitting on whitespace, however, and Chapter 5 introduces more sophisticated approaches for tokenization that work better for the general case.

Example 4-6. Hacking on Google Plus data in the interpreter with NLTK.

```
>>> import nltk
>>> import json
>>> data = json.loads(open("107033731246200681024.plus").read())
>>> all_content = " ".join([ a['object']['content'] for a in data ])
>>> len(all_content)
203026
>>> tokens = all_content.split()
>>> text = nltk.Text(tokens)
>>> text.concordance("open")
Building index...
Displaying 12 of 12 matches:
 . Part of what matters so much when open source, the web, and open data meet
 much when open source, the web, and open data meet government is that practic
 government to provide a platform to open and share freely downloadable GIS da
s bill to prevent NIH from mandating open access to federally funded research.
 agencies such as the NIH to mandate open access to research that is funded by
```

```
lly like + John Tolva 's piece about open government in Chicago. It ties toget
orce. Analysis builds new processes. open data builds businesses." I haven't s
e in. We wanted reading to remain as open as it did when printed books ruled t
 a stand for reading portability and open access. Offer books in multiple open
open access. Offer books in multiple open formats, DRM-free, and from multiple
hat story redefined free software as open source, and the world hasn't been th
g behind them. Thinking deeply about open source and the internet got me think
>>> text.collocations()
Building collocations list
AlhambraPatternsAndTextures AlhambraPatternsAndTextures;
OccupyWallStreet OccupyWallStreet; PopupCrecheShops PopupCrecheShops;
Wall Street; Maker Faire; Mini Maker; New York; Bay Mini; East Bay;
United States; Ada Lovelace; hedge fund; Faire East; Steve Jobs; 21st
century; Silicon Valley; sales tax; Jennifer Pahlka; Mike Loukides;
data science
>>> fdist = text.vocab()
Building vocabulary index...
>>> fdist["open"]
11
>>> fdist["source"]
5
>>> fdist["web"]
6
>>> fdist["2.0"]
4
>>> len(tokens)
33280
>>> len(fdist.keys()) # unique tokens
8598
>>> [w for w in fdist.keys()[:100] \
... if w.lower() not in nltk.corpus.stopwords.words('english')]
[u'one', u'-', u'new', u'+', u'like', u'government', u'would', u'people', u'get',
 u'data', u'make', u'really', u'also', u'many', u"It's", u'business', u"I'm", u"it's",
 u'much', u'way', u'AlhambraPatternsAndTextures', u'book', u'see', u'work', u'great',
 u'market']>>> [w for w in fdist.keys() if len(w) > 15 and not w.startswith("http")]
[u'one', u'-', u'new', u'+', u'like', u'government', u'would', u'people', u'get', u'data',
 u'make', u'really', u'also', u'many', u"It's", u'business', u"I'm", u"it's", u'much',
 u'way', u'AlhambraPatternsAndTextures', u'book', u'see', u'work', u'great', u'market']
>>> [w for w in fdist.keys() if len(w) > 15 and not w.startswith("http")]
[u'AlhambraPatternsAndTextures', u'OccupyWallStreet', u'PopupCrecheShops',
 u'#OccupyWallStreet', u'"CurrentCapacity"', u'"DesignCapacity"', u'"LegacyBatteryInfo"',
 u'"unconferences,"', u'@OReillyMedia#Ebook', u'@josephjesposito', u'@netgarden(come-on,',
 u'@oreillymedia"deal', u'Administration\u2014which', u'Amazon-California',
 u'Community-generated', u'Frisch(12:00-1:00)',u'Internet-related', u'Representatives.[10]',
 u'Republican-controlled', u'Twitter,@_vrajesh', u"administration's", u'all-encompassing',
 u'augmented-reality', u'books-to-diapers-to-machetes', u'brain-wave-sensing',
 u'brick-and-mortar', u'codeforamerica.org', u'continuously-updated', u'crimespotting.org',
 u'disobedience\u2014including', u'dolphin-slaughter', u'community-driven',
 u'connect-the-dots', u'entertainment-sector', u'government-funded', u'informationdiet.com',
 u'instantaneously.', u'little-mentioned', u'micro-entrepreneurship.', u'nineteenth-century',
 u'opsociety.org/securedonation.htm', u'problem/solution/explanation', u'problematically,',
 u'reconceptualization', u'responsibility."',u'scientifically-literate', u'self-employment,',
```

```
 u'subscription-based', u'technology-based', u'transportation.)', u'two-dimensional,',
 u'unintentionally?', u'watercolor-style',
 u'{"Amperage"=18446744073709548064,"Flags"=4,"Capacity"=4464,"Current"=2850,"Voltage"=7362,
 "Cycle', u'\u201cverbivocovisual\u201d']
>>> len([w for w in fdist.keys() if w.startswith("http")])
98
>>> for rank, word in enumerate(fdist): print rank, word, fdist[word]
0 the 1815
1 of 1017
2 to 988
3 and 701
4 a 699
5 in 546
6 that 535
7 is 413
... output truncated ...
```



You may need to run `nltk.download('stopwords')` to download NLTK's stopwords data if you haven't already installed it. If possible, it is recommended that you just run `nltk.download()` to install all of the NLTK data.

The last command in the interpreter session lists the words from the frequency distribution, sorted by frequency. Not surprisingly, common words like "the," "to," and "of" —stopwords—are the most frequently occurring, but there's a steep decline and the distribution has a very long tail. We're working with a small sample of text data, but this same property will hold true for any frequency analysis of natural language. Zipf's law, a well-known empirical law of natural language, asserts that a word's frequency within a corpus is inversely proportional to its rank in the frequency table. What this means is that if the most frequently occurring term in a corpus accounts for N% of the total words, the second most frequently occurring term in the corpus should account for (N/2)% of the words, the third most frequent term for (N/3)% of the words, etc. When graphed, such a distribution (even for a small sample of data) shows a curve that hugs each axis, as you can see in Figure 4-3. An important observation is that most of the area in such a distribution lies in its tail, which for a corpus large enough to span a reasonable sample of a language is always quite long. If you were to plot this kind of distribution on a chart where each axis was scaled by a logarithm, the curve would approach a straight line for a representative sample size.

Figure 4-3. The frequency distribution for terms appearing in a small sample of Google + data "hugs" each axis

Zipf's law gives you insight into what a frequency distribution for words appearing in a corpus should look like, and it provides some rules of thumb that can be useful in estimating frequency. For example, if you know that there are a million (non-unique) words in a corpus, and you assume that the most frequently used word (usually "the," in English) accounts for 7% of the words,[2] you could derive the total number of logical calculations an algorithm performs if you were to consider a particular slice of the terms from the frequency distribution. Sometimes, this kind of simple arithmetic on the back of a napkin is all that it takes to sanity-check assumptions about a long-running wall-clock time, or confirm whether certain computations on a large enough data set are even tractable.

Can you graph the same kind of curve shown in Figure 4-3 for the content from a few hundred Google+ activities using the techniques introduced in this chapter in combination with IPython's plotting functionality that was introduced in Chapter 1?

2. The word "the" accounts for 7% of the tokens in the Brown Corpus and provides a reasonable starting point for a corpus if you don't know anything else about it.

A Whiz-Bang Introduction to TF-IDF

Although rigorous approaches to natural language processing (NLP) that include such things as sentence segmentation, tokenization, word chunking, and entity detection are necessary in order to achieve the deepest possible understanding of textual data, it's helpful to first introduce some fundamentals from Information Retrieval theory. The remainder of this chapter introduces some of its more foundational aspects, including TF-IDF, the cosine similarity metric, and some of the theory behind collocation detection. Chapter 5 provides a deeper discussion of NLP as a logical continuation of this discussion.

If you want to dig deeper into IR theory, the full text of *Introduction to Information Retrieval* is available online and provides more information than you could (probably) ever want to know about the field.

Information retrieval is an extensive field with many specialties. This discussion narrows in on TF-IDF, one of the most fundamental techniques for retrieving relevant documents from a corpus. TF-IDF stands for term frequency-inverse document frequency and can be used to query a corpus by calculating normalized scores that express the relative importance of terms in the documents. Mathematically, TF-IDF is expressed as the product of the term frequency and the inverse document frequency, *tf_idf = tf*idf*, where the term `tf` represents the importance of a term in a specific document, and `idf` represents the importance of a term relative to the entire corpus. Multiplying these terms together produces a score that accounts for both factors and has been an integral part of every major search engine at some point in its existence. To get a more intuitive idea of how TF-IDF works, let's walk through each of the calculations involved in computing the overall score.

For simplicity in illustration, suppose you have a corpus containing three sample documents and terms are calculated by simply breaking on whitespace, as illustrated in Example 4-7.

Example 4-7. Sample data structures used in illustrations throughout this chapter

```
corpus = {
        'a' : "Mr. Green killed Colonel Mustard in the study with the candlestick. \
Mr. Green is not a very nice fellow.",
        'b' : "Professor Plumb has a green plant in his study.",
        'c' : "Miss Scarlett watered Professor Plumb's green plant while he was away \
from his office last week."
}

terms = {
        'a' : [ i.lower() for i in corpus['a'].split() ],
        'b' : [ i.lower() for i in corpus['b'].split() ],
```

```
'c' : [ i.lower() for i in corpus['c'].split() ]
}
```

A term's frequency could simply be represented as the number of times it occurs in the text, but it is more commonly the case that it is normalized by taking into account the total number of terms in the text, so that the overall score accounts for document length relative to a term's frequency. For example, the term "green" (once normalized to lowercase) occurs twice in `corpus['a']` and only once in `corpus['b']`, so `corpus['a']` would produce a higher score if frequency were the only scoring criterion. However, if you normalize for document length, `corpus['b']` would have a slightly higher term frequency score for "green" (1/9) than `corpus['a']` (2/19), because `corpus['b']` is shorter than `corpus['a']`—even though "green" occurs more frequently in `corpus['a']`. Thus, a common technique for scoring a compound query such as "Mr. Green" is to sum the term frequency scores for each of the query terms in each document, and return the documents ranked by the summed term frequency score.

The previous paragraph is actually not as confusing as it sounds; for example, querying our sample corpus for "Mr. Green" would return the normalized scores reported in Table 4-1 for each document.

Table 4-1. Sample term frequency scores for "Mr. Green"

Document	tf(Mr.)	tf(Green)	Sum
corpus['a']	2/19	2/19	4/19 (0.2105)
corpus['b']	0	1/9	1/9 (0.1111)
corpus['c']	0	1/16	1/16 (0.0625)

For this contrived example, a cumulative term frequency scoring scheme works out and returns `corpus['a']` (the document that we'd expect it to return), since `corpus['a']` is the only one that contains the compound token "Mr. Green". However, a number of problems could have emerged because the term frequency scoring model looks at each document as an unordered collection of words. For example, queries for "Green Mr." or "Green Mr. Foo" would have returned the exact same scores as the query for "Mr. Green", even though neither of those compound tokens appear in the sample sentences. Additionally, there are a number of scenarios that we could easily contrive to illustrate fairly poor results from the term frequency ranking technique by exploiting the fact that trailing punctuation is not properly handled, and the context around tokens of interest is not taken into account by the calculations.

Considering term frequency alone turns out to be a common problem when scoring on a document-by-document basis because it doesn't account for very frequent words that are common across many documents. Meaning, all terms are weighted equally regardless of their actual importance. For example, "the green plant" contains the stopword "the", which skews overall term frequency scores in favor of `corpus['a']` because "the"

appears twice in that document, as does "green". In contrast, in `corpus['c']` "green" and "plant" each appear only once.

Consequently, the scores would break down as shown in Table 4-2, with `corpus['a']` ranked as more relevant than `corpus['c']` even though intuition might lead you to believe that ideal query results probably shouldn't have turned out that way. (Fortunately, however, `corpus['b']` still ranks highest.)

Table 4-2. Sample term frequency scores for "the green plant"

Document	tf(the)	tf(green)	tf(plant)	Sum
corpus['a']	2/19	2/19	0	4/19 (0.2105)
corpus['b']	0	1/9	1/9	2/9 (0.2222)
corpus['c']	0	1/16	1/16	1/8 (0.125)

Toolkits such as NLTK provide lists of stopwords that can be used to filter out terms such as "the", "a", "and", etc., but keep in mind that there may be terms that evade even the best stopword lists and yet still are quite common to specialized domains. The inverse document frequency metric is a calculation that provides a generic normalization metric for a corpus, and accounts for the appearance of common terms across a set of documents by taking into consideration the total number of documents in which a query term ever appears. The intuition behind this metric is that it produces a higher value if a term is somewhat uncommon across the corpus than if it is very common, which helps to account for the problem with stopwords we just investigated. For example, a query for "green" in the corpus of sample documents should return a lower inverse document frequency score than "candlestick", because "green" appears in every document while "candlestick" appears in only one. Mathematically, the only nuance of interest for the inverse document frequency calculation is that a logarithm is used to squash the result into a compressed range (as shown in Figure 4-4) since its usual application is in multiplying it against term frequency as a scaling factor.

At this point, we've come full circle and devised a way to compute a score for a multiterm query that accounts for the frequency of terms appearing in a document, the length of the document in which any particular term appears, and the overall uniqueness of the terms across documents in the entire corpus. In other words, `tf-idf = tf*idf`. Example 4-8 is a naive implementation of this discussion that should help solidify the concepts described. Take a moment to review it, and then we'll discuss a few sample queries.

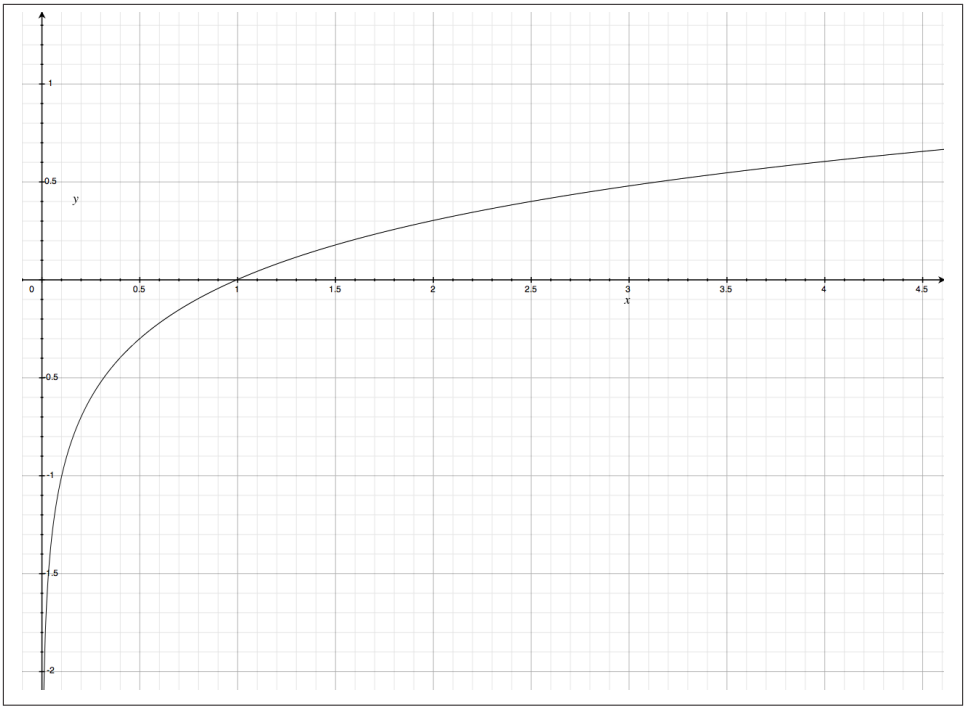


Figure 4-4. The logarithm squashes a large range of values into a more compressed space

Example 4-8. Running TF-IDF on sample data

```
from math import log

# XXX: Enter in a query term from the corpus variable
QUERY_TERMS = ['green', 'mustard']

def tf(term, doc, normalize=True):
    doc = doc.lower().split()
    if normalize:
        return doc.count(term.lower()) / float(len(doc))
    else:
        return doc.count(term.lower()) / 1.0


def idf(term, corpus):
    num_texts_with_term = len([True for text in corpus if term.lower()
                               in text.lower().split()])

    # tf-idf calc involves multiplying against a tf value less than 0, so it's important
    # to return a value greater than 1 for consistent scoring. (Multiplying two values
    # less than 1 returns a value less than each of them)
```

```
    try:
        return 1.0 + log(float(len(corpus)) / num_texts_with_term)
    except ZeroDivisionError:
        return 1.0


def tf_idf(term, doc, corpus):
    return tf(term, doc) * idf(term, corpus)


corpus = \
    {'a': 'Mr. Green killed Colonel Mustard in the study with the candlestick. \
Mr. Green is not a very nice fellow.',
     'b': 'Professor Plumb has a green plant in his study.',
     'c': "Miss Scarlett watered Professor Plumb's green plant while he was away \
from his office last week."}

for (k, v) in sorted(corpus.items()):
    print k, ':', v
print

# Score queries by calculating cumulative tf_idf score for each term in query

query_scores = {'a': 0, 'b': 0, 'c': 0}
for term in [t.lower() for t in QUERY_TERMS]:
    for doc in sorted(corpus):
        print 'TF(%s): %s' % (doc, term), tf(term, corpus[doc])
    print 'IDF: %s' % (term, ), idf(term, corpus.values())
    print

    for doc in sorted(corpus):
        score = tf_idf(term, corpus[doc], corpus.values())
        print 'TF-IDF(%s): %s' % (doc, term), score
        query_scores[doc] += score
    print

print "Overall TF-IDF scores for query '%s'" % (' '.join(QUERY_TERMS), )
for (doc, score) in sorted(query_scores.items()):
    print doc, score
```

Sample output follows:

```
    a : Mr. Green killed Colonel Mustard in the study...
    b : Professor Plumb has a green plant in his study.
    c : Miss Scarlett watered Professor Plumb's green...

    TF(a): mr. 0.105263157895
    TF(b): mr. 0.0
    TF(c): mr. 0.0
    IDF: mr. 2.09861228867

    TF-IDF(a): mr. 0.220906556702
```

```
TF-IDF(b): mr. 0.0
TF-IDF(c): mr. 0.0

TF(a): green 0.105263157895
TF(b): green 0.111111111111
TF(c): green 0.0625
IDF: green 1.0

TF-IDF(a): green 0.105263157895
TF-IDF(b): green 0.111111111111
TF-IDF(c): green 0.0625

Overall TF-IDF scores for query 'mr. green'
a 0.326169714597
b 0.111111111111
c 0.0625
```

Although we're working on a trivially small scale, the calculations involved work the same for larger data sets. Table 4-3 is a consolidated adaptation of the program's output for three sample queries that involve four distinct terms:

- "green"
- "Mr. Green"
- "the green plant"

Even though the IDF calculations for terms are for the entire corpus, they are displayed on a per-document basis so that you can easily verify TF-IDF scores by skimming a single row and multiplying two numbers. Again, it's worth taking a few minutes to understand the data in the table so that you have a good feel for the way the calculations work. It's remarkable just how powerful TF-IDF is, given that it doesn't account for the proximity or ordering of words in a document.

Table 4-3. Calculations involved in TF-IDF sample queries, as computed by Example 4-8

Document	tf(mr.)	tf(green)	tf(the)	tf(plant)
corpus['a']	0.1053	0.1053	1.1053	0
corpus['b']	0	0.1111	0	0.1111
corpus['c']	0	0.0625	0	0.0625

	idf(mr.)	idf(green)	idf(the)	idf(plant)
corpus['a']	2.0986	1.0	2.099	1.4055
corpus['b']	2.0986	1.0	2.099	1.4055
corpus['c']	2.0986	1.0	2.099	1.4055

	tf-idf(mr.)	tf-idf(green)	tf-idf(the)	tf-idf(plant)
corpus['a']	0.2209	0.1053	0.2209	0
corpus['b']	0	0.1111	0	0.1562
corpus['c']	0	0.0625	0	0.0878

The same results for each query are shown in Table 4-4, with the TF-IDF values summed on a per-document basis.

Table 4-4. Summed TF-IDF values for sample queries as computed by Example 4-8

Query	corpus['a']	corpus['b']	corpus['c']
green	0.1053	**0.1111**	0.0625
Mr. Green	0.2209 + 0.1053 = **0.3262**	0 + 0.1111 = 0.1111	0 + 0.0625 = 0.0625
the green plant	0.2209 + 0.1053 + 0 = **0.3262**	0 + 0.1111 + 0.1562 = 0.2673	0 + 0.0625 + 0.0878 = 0.1503

From a qualitative standpoint, the query results seem reasonable. The `corpus['b']` document is the winner for the query "green", with `corpus['a']` just a hair behind. In this case, the deciding factor was the length of `corpus['b']` being much smaller than `corpus['a']`: the normalized TF score tipped in favor of `corpus['b']` for its one occurrence of "green", even though "Green" appeared in `corpus['a']` two times. Since "green" appears in all three documents, the net effect of the IDF term in the calculations was a wash. Do note, however, that if we had returned 0.0 instead of 1.0 for IDF calculations, as is done in some implementations, the TF-IDF scores for "green" would have been 0.0 for all three documents. Depending on the particular situation, it may be better to return 0.0 for the IDF scores rather than 1.0. For example, if you had 100,000 documents and "green" appeared in all of them, you'd almost certainly consider it to be a stopword and want to remove its effects in a query entirely.

A very worthwhile exercise to consider is why `corpus['a']` scored highest for "the green plant" as opposed to `corpus['b']`, which at first blush, might have seemed a little more obvious.

A finer point to observe is that the sample implementation provided in Example 4-8 adjusts the IDF score by adding a value of 1.0 to the logarithm calculation, for the purposes of illustration and because we're dealing with a trivial document set. Without the 1.0 adjustment in the calculation, it would be possible to have the `idf` function return values that are less than 1.0, which would result in two fractions being multiplied in the TF-IDF calculation. Since multiplying two fractions together results in a value smaller than either of them, this turns out to be an easily overlooked edge case in the TF-IDF calculation. Recall that the intuition behind the TF-IDF calculation is that we'd like to be able to multiply two terms in a way that consistently produces larger TF-IDF scores for more-relevant queries than for less-relevant queries.

Querying Google+ Data with TF-IDF

Let's apply TF-IDF to the Google+ data we collected earlier and see how it works out as a tool for querying the data. NLTK provides some abstractions that we can use instead of rolling our own, so there's actually very little to do now that you understand the underlying theory. The listing in Example 4-9 assumes you saved the working Google + data from earlier in this chapter as a JSON file, and it allows you to pass in multiple query terms that are used to score the documents by relevance.

Example 4-9. Querying Google+ data with TF-IDF

```
import json
import nltk

# Load in unstructured data from wherever you've saved it

DATA = '10703731246200681024.json'
data = json.loads(open(DATA).read())

# XXX: provide your own query terms here

QUERY_TERMS = ['open', 'healthcare']

activities = [activity['object']['content'].lower().split() \
              for activity in data \
                if activity['object']['content'] != ""]

# TextCollection provides tf, idf, and tf_idf abstractions so
# that we don't have to maintain/compute them ourselves

tc = nltk.TextCollection(activities)

relevant_activities = []

for idx in range(len(activities)):
    score = 0
    for term in [t.lower() for t in QUERY_TERMS]:
        score += tc.tf_idf(term, activities[idx])
    if score > 0:
        relevant_activities.append({'score': score, 'title': data[idx]['title'],
                                    'url': data[idx]['url']})

# Sort by score and display results

relevant_activities = sorted(relevant_activities, key=lambda p: p['score'], reverse=True)
for activity in relevant_activities:
    print activity['title']
    print '\tLink: %s' % (activity['url'], )
    print '\tScore: %s' % (activity['score'], )
    print
```

Sample query results for "SOPA", a controversial piece of proposed legislation, on Tim O'Reilly's Google+ data are shown in Example 4-10.

Example 4-10. Sample results from Example 4-9

```
I think the key point of this piece by +Mike Loukides, that PIPA and SOPA provide a
"right of ext...
    Link: https://plus.google.com/107033731246200681024/posts/ULi4RYpvQGT
    Score: 0.0805961208217

Learn to Be a Better Activist During the SOPA Blackouts +Clay Johnson has put together
an awesome...
    Link: https://plus.google.com/107033731246200681024/posts/hrC5aj7gS6v
    Score: 0.0255051015259

SOPA and PIPA are bad industrial policy There are many arguments against SOPA and PIPA
that are b...
    Link: https://plus.google.com/107033731246200681024/posts/LZs8TekXK2T
    Score: 0.0227351539694

Further thoughts on SOPA, and why Congress shouldn't listen to lobbyists Colleen Taylor
of GigaOM...
    Link: https://plus.google.com/107033731246200681024/posts/5Xd3VjFR8gx
    Score: 0.0112879721039

...
```

Given a search term, being able to narrow in on three Google+ content items ranked by relevance is of tremendous benefit when analyzing unstructured text data. Try out some other queries and qualitatively review the results to see for yourself how well the TF-IDF metric works, keeping in mind that the absolute values of the scores aren't really important—it's the ability to find and sort documents by relevance that matters. Then, begin to ponder the gazillion ways that you could tune or augment this metric to be even more effective. (After all, every data hacker has to complete this exercise *at least* one time in life. Whether you have to complete it a second or third time depends on whether or not you forgot what happened the first time you did it.) One obvious improvement that's left as an exercise for the reader is to stem verbs so that variations in tense, grammatical role, etc., resolve to the same stem and can be more accurately accounted for in similarity calculations. The `nltk.stem` module provides easy-to-use implementations for several common stemming algorithms.

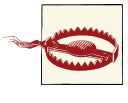
Let's take our new tools and apply them to the foundational problem of computing similar documents. After all, once you've narrowed in on a document of interest, the next natural step is to discover other content that might be of interesting.

Finding Similar Documents

Once you've queried and discovered documents of interest, one of the next things you might want to do is find similar documents. Whereas TF-IDF can provide the means to narrow down a corpus based on search terms, cosine similarity is one of the most common techniques for comparing documents to one another, which is the essence of finding a similar document. An understanding of cosine similarity requires a brief introduction to vector space models, which is the topic of the next section.

The Theory Behind Vector Space Models and Cosine Similarity

While it has been emphasized that TF-IDF models documents as unordered collections of words, another convenient way to model documents is with a model called a vector space. The basic theory behind a vector space model is that you have a large multidimensional space that contains one vector for each document, and the distance between any two vectors indicates the similarity of the corresponding documents. One of the most beautiful things about vector space models is that you can also represent a query as a vector and find the most relevant documents for the query by finding the document vectors with the shortest distance to the query vector. Although it's virtually impossible to do this subject justice in a short section, it's important to have a basic understanding of vector space models if you have any interest at all in text mining or the IR field. If you're not interested in the background theory and want to jump straight into implementation details on good faith, feel free to skip ahead to the next section.



This section assumes a basic understanding of trigonometry. If your trigonometry skills are a little rusty, consider this section a great opportunity to brush up on high school math. Or, if you're not feeling up to it, just skim this section and rest assured that thre is some mathematical rigor that backs the similarity computation that we'll be employing to find similar documents.

First, it might be helpful to clarify exactly what is meant by the term "vector," since there are so many subtle variations associated with it across various fields of study. Generally speaking, a vector is a list of numbers that expresses both a direction relative to an origin and a magnitude, which is the distance from that origin. A vector can very naturally be represented as a line segment between the origin and a point in an N-dimensional space by drawing a line between the origin and the point. To illustrate, imagine a document that is defined by only two terms ("Open", "Web"), with a corresponding vector of (*0.45*, *0.67*), where the values in the vector are values such as TF-IDF scores for the terms. In a vector space, this document could be represented in two dimensions by a line segment extending from the origin at (*0,0*) to the point at (*0.45, 0.67*). In reference to an x/y plane, the x-axis would represent "Open", the y-axis would represent "Web", and the

vector from (*0,0*) to (*0.45, 0.67*) would represent the document in question. However, interesting documents generally contain hundreds of terms at a minimum, but the same fundamentals apply for modeling documents in these higher-dimensional spaces; it's just harder to visualize.

Try making the transition from visualizing a document represented by a vector with two components to a document represented by three dimensions, such as ("Open", "Web", "Government"). Then consider taking a leap of faith and accepting that although it's hard to visualize, it is still possible to have a vector represent additional dimensions that you can't easily sketch out or see. If you're able to do that, you should have no problem believing the same vector operations that can be applied to a 2-dimensional space can be equally as well applied to a 10-dimensional space or a 367-dimensional space. Figure 4-5 shows an example vector in 3-dimensional space.

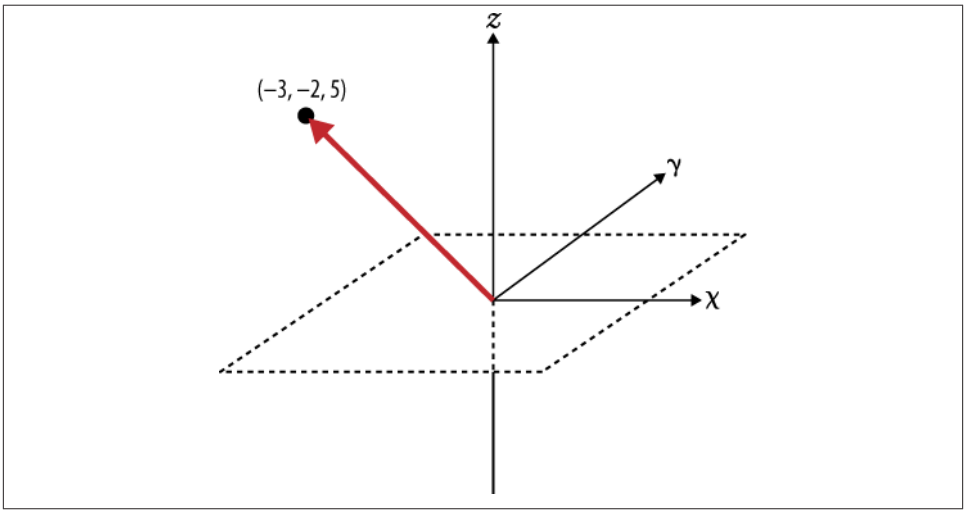


Figure 4-5. An example vector with the value (–3, –2, 5) plotted in 3D space

Given that it's possible to model documents as term-centric vectors, with each term in the document represented by its corresponding TF-IDF score, the task then is to determine what metric best represents the similarity between two documents. As it turns out, the cosine of the angle between any two vectors is a valid metric for comparing them and is known as the cosine similarity of the vectors. Although perhaps not yet intuitive, years of scientific research have demonstrated that computing the cosine similarity of documents represented as term vectors is a very effective metric. (It does suffer from many of the same problems as TF-IDF, though; see "Closing Remarks" on page 162 for a very brief synopsis.) Building up a rigorous proof of the details behind the cosine similarity metric would be beyond the scope of this book, but the gist is that the cosine of the angle between any two vectors indicates the similarity between them and is

equivalent to the dot product of their unit vectors. Intuitively, it might be helpful to consider that the closer two vectors are to one another, the smaller the angle between them will be, and thus the larger the cosine of the angle between them will be. Two identical vectors would have an angle of 0 degrees and a similarity metric of 1.0, while two vectors that are orthogonal to one another would have an angle of 90 degrees and a similarity metric of 0.0. The following sketch attempts to demonstrate:

$\vec{doc1} \cdot \vec{doc2} = \|doc1\| \cdot \|doc2\| \cdot \cos\Theta$	Given (by trigonometry)
$\frac{\vec{doc1} \cdot \vec{doc2}}{\|doc1\| \cdot \|doc2\|} = \cos\Theta$	By division
$\hat{doc1} \cdot \hat{doc2} = \cos\Theta$	By definition of 'unit vector'
$\hat{doc1} \cdot \hat{doc2} = \text{Similarity (doc1, doc2)}$	By substitution (assume: $\cos\Theta = \text{Similarity (doc1, doc2)}$

XXX: PRODUCTION: the previous figure has a slight error. The bottom cell is actually two cells and should be split as such so that there are 4 total cells with alternating colors in each row.

Recalling that a unit vector has a length of 1.0 (by definition), the beauty of computing document similarity with unit vectors is that they're already normalized against what might be substantial variations in length. Given that brief math lesson, you're probably ready to see some of this in action. That's what the next section is all about.

Clustering Posts with Cosine Similarity

If you don't care to remember anything else from the previous section, just remember this: *to compute the similarity between two documents, you really just need to produce a term vector for each document and compute the dot product of the unit vectors for those documents.* Conveniently, NLTK exposes the `nltk.cluster.util.cosine_distance(v1,v2)` function for computing cosine similarity, so it really is pretty straightforward to compare documents. As Example 4-11 shows, all of the work involved is in producing the appropriate term vectors; in short, it computes term vectors for a given pair of documents by assigning TF-IDF scores to each component in the vectors. Because the exact vocabularies of the two documents are probably not identical, however, placeholders with a value of 0.0 must be left in each vector for words that are missing from the document at hand but present in the other one. The net effect is that you end up with two vectors of identical length with components ordered identically that can be used to perform the vector operations.

For example, suppose document1 contained the terms (A, B, C) and had the corresponding vector of TF-IDF weights (0.10, 0.15, 0.12), while document2 contained the terms (C, D, E) with the corresponding vector of TF-IDF weights (0.05, 0.10, 0.09). The derived vector for document1 would be (0.10, 0.15, 0.12, 0.0, 0.0), and the derived vector for document2 would be (0.0, 0.0, 0.05, 0.10, 0.09). Each of these vectors could be passed into NLTK's `cosine_distance` function, which yields the cosine similarity. Internally, `cosine_distance` uses the `numpy` module to very efficiently compute the dot product of the unit vectors, and that's the result. Although the code in this section reuses the TF-IDF calculations that were introduced previously, the exact scoring function could be any useful metric. TF-IDF (or some variation thereof), however, is quite common for many implementations and provides a great starting point.

Example 4-11 illustrates an approach for using cosine similarity to find the most similar document to each document in a corpus of Google+ data. It should apply equally as well to any other type of unstructured data, such as blog posts, books, etc.

Example 4-11. Finding similar documents using cosine similarity

```
# -*- coding: utf-8 -*-

import sys
import json
import nltk

# Load in textual data from wherever you've saved it

DATA = sys.argv[1]

# Only consider content that's ~100+ words.
data = [post for post in json.loads(open(DATA).read())
        if len(post['object']['content']) > 1000]

all_posts = [post['object']['content'].lower().split()
             for post in data ]

# Provides tf/idf/tf_idf abstractions for scoring

tc = nltk.TextCollection(all_posts)

# Compute a term-document matrix such that td_matrix[doc_title][term]
# returns a tf-idf score for the term in the document

td_matrix = {}
for idx in range(len(all_posts)):
    post = all_posts[idx]
    fdist = nltk.FreqDist(post)

    doc_title = data[idx]['title']
    url = data[idx]['url']
    td_matrix[(doc_title, url)] = {}
```

```
    for term in fdist.iterkeys():
        td_matrix[(doc_title, url)][term] = tc.tf_idf(term, post)

# Build vectors such that term scores are in the same positions...

distances = {}
for (title1, url1) in td_matrix.keys():

    distances[(title1, url1)] = {}
    (min_dist, most_similar) = (0.0, ('', ''))

    for (title2, url2) in td_matrix.keys():

        # Take care not to mutate the original data structures
        # since we're in a loop and need the originals multiple times

        terms1 = td_matrix[(title1, url1)].copy()
        terms2 = td_matrix[(title2, url2)].copy()

        # Fill in "gaps" in each map so vectors of the same length can be computed

        for term1 in terms1:
            if term1 not in terms2:
                terms2[term1] = 0

        for term2 in terms2:
            if term2 not in terms1:
                terms1[term2] = 0

        # Create vectors from term maps

        v1 = [score for (term, score) in sorted(terms1.items())]
        v2 = [score for (term, score) in sorted(terms2.items())]

        # Compute similarity amongst documents

        distances[(title1, url1)][(title2, url2)] = \
            nltk.cluster.util.cosine_distance(v1, v2)

        if url1 == url2:
            continue

        if distances[(title1, url1)][(title2, url2)] > min_dist:
            (min_dist, most_similar) = (distances[(title1, url1)][(title2,
                                        url2)], (title2, url2))

    print '''Most similar to %s (%s)
\t%s (%s)
\tscore %d
''' % (title1, url1,
            most_similar[0], most_similar[1], 1-min_dist)
```

If you've found this discussion of cosine similarity interesting, it might at first seem almost magical when you realize that *the best part is that querying a vector space is the very same operation as computing the similarity between documents, except that instead of comparing just document vectors, you compare your query vector and the document vectors.* Take a moment to think about it: it's a rather profound insight that the mathematics work out that way.

In terms of implementing a program to compute similarity across an entire corpus, however, take note that the naive approach means constructing a vector containing your query terms and comparing it to every single document in the corpus. Clearly, the approach of directly comparing a query vector to every possible document vector is not a good idea for even a corpus of modest size, and you'd need to make some very good engineering decisions involving the appropriate use of indexes to achieve a scalable solution. We briefly touched upon the fundamental problem of needing a dimensionality reduction as a common staple in clustering in Chapter 3, and here we see the same concept emerge. *Anytime you encounter a similarity computation, you will almost imminently encounter the need for a dimensionality reduction to make the computation tractable.*

Visualizing Document Similarity with a Matrix Diagram

Like just about everything else in this book, there's certainly more than one way to visualize the similarity between items. The approach introduced in this section is to use graph-like structures, where a link between documents encodes a measure of the similarity between them. This situation presents an excellent opportunity to introduce more visualizations from D3, a state-of-the-art visualization toolkit. D3 is specifically designed with the interests of data scientists in mind, offers a familiar declarative syntax, and achieves a nice middle ground between high-level and low-level interfaces. A minimal (uninteresting) adaptation to Example 4-11 is all that's needed to emit a collection of nodes and edges that can be used to produce visualizations similar to those in the D3 examples gallery. A nested loop can compute the similarity between the working sample of Google+ data from this chapter, and linkages between items may be determined based upon a simple statistical thresholding criterion.

The details associated with munging the data and producing the output required to power the visualizations won't be presented here, but the turn-key example code is provided in the IPython Notebook for this chapter.

The code produces the matrix diagram in Figure 4-6. An advantage of matrix diagram versus a graph-based layout is that there's no potential for messy overlap between edges that represent linkages, so you avoid the proverbial "hairball" problem with your display.

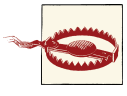
However, the ordering of rows and columns affects the intution about the patterns that may be present in the matrix, so careful thought should be placed into the best ordering for the rows and columns. Usually, rows and columns have additional properties that could be used to order them in order to make it easier to pinpoint patterns in the data. A recommended exercise for this chapter is to spend some time enhancing the capabilities of this matrix diagram.

Figure 4-6. A matrix diagram displaying linkages between Google+ activities

Bigram Analysis

As previously mentioned, one issue that is frequently overlooked in unstructured text processing is the tremendous amount of information gained when you're able to look at more than one token at a time, because so many concepts we express are phrases and not just single words. For example, if someone were to tell you that a few of the most common terms in a post are "open", "source", and "government", could you necessarily say that the text is probably about "open source", "open government", both, or neither? If you had a priori knowledge of the author or content, you could probably make a good guess, but if you were relying totally on a machine to try to classify the nature of a document as being about collaborative software development or transformational government, you'd need to go back to the text and somehow determine which of the words most frequently occur after "open"—i.e., you'd like to find the *collocations* that start with the token "open".

Recall from Chapter 3 that an *n*-gram is just a terse way of expressing each possible consecutive sequence of *n* tokens from a text, and it provides the foundational data structure for computing collocations. There are always (*n*-1) *n*-grams for any value of *n*, and if you were to consider all of the bigrams (2-grams) for the sequence of tokens `["Mr.", "Green", "killed", "Colonel", "Mustard"]`, you'd have four possibilities: `[("Mr.", "Green"), ("Green", "killed"), ("killed", "Colonel"), ("Colonel", "Mustard")]`. You'd need a larger sample of text than just our sample sentence to determine collocations, but assuming you had background knowledge or additional text, the next step would be to statistically analyze the bigrams in order to determine which of them are likely to be collocations.



It's worth noting that the storage necessary for persisting an *n*-gram model requires space for (*T*–1)**n* tokens (which is practically *T***n*), where *T* is the number of tokens in question and *n* is defined by the size of the desired *n*-gram. As an example, assume a document contains 1,000 tokens and requires around 8 KB of storage. Storing all bigrams for the text would require roughly double the original storage, or 16 KB, as you would be storing 999*2 tokens plus overhead. Storing all trigrams for the text (998*3 tokens plus overhead) would require roughly triple the original storage, or 24 KB. Thus, without devising specialized data structures or compression schemes, the storage costs for *n*-grams can be estimated as *n* times the original storage requirement for any value of *n*.

n-grams are very simple yet very powerful as a technique for clustering commonly co-occurring words. If you compute all of the *n*-grams for even a small value of *n*, you're likely to discover that some interesting patterns emerge from the text itself with no additional work required. For example, in considering the bigrams for a sufficiently long text, you're likely to discover the proper names, such as "Mr. Green" and "Colonel Mustard", concepts such as "open source" or "open government", and so forth. Similar patterns emerge when you consider frequent trigrams and *n*-grams for values of *n* slightly larger than three. In fact, computing bigrams in this way produces essentially the same results as the `collocations` function that you ran in an earlier interpreter session, except that some additional statistical analysis takes into account the use of rare words. As you already know from the interpreter session we looked at earlier in this chapter (Example 4-6), NLTK takes care of most of the effort in computing *n*-grams, discovering collocations from text, discovering the context in which a token has been used, etc. Example 4-12 demonstrates.

Example 4-12. Using NLTK to compute bigrams and collocations for a sentence

```
>>> import nltk
>>> nltk.ngrams("Mr. Green killed Colonel Mustard in the study with the \
... candlestick. Mr. Green is not a very nice fellow.".split(), 2)
```

```
[('Mr.', 'Green'), ('Green', 'killed'), ('killed', 'Colonel'),
 ('Colonel', 'Mustard'), ('Mustard', 'in'), ('in', 'the'),
 ('the', 'study'), ('study', 'with'), ('with', 'the'),
 ('the', 'candlestick.'), ('candlestick.', 'Mr.'), ('Mr.', 'Green'),
 ('Green', 'is'), ('is', 'not'), ('not', 'a'), ('a', 'very'),
 ('very', 'nice'), ('nice', 'fellow.')]
>>> txt = nltk.Text("Mr. Green killed Colonel Mustard in the study with the\
... candletick. Mr. Green is not a very nice fellow.".split())
>>> txt.collocations()
Building collocations list
Mr. Green
```

Recall that the only drawback to using built-in demo functionality such as `nltk.Text.collocations` is that these functions don't return data structures that you can store and manipulate. Whenever you run into such a situation, just take a look at the underlying source code, which is usually pretty easy to adapt for your own purposes. Example 4-13 illustrates how you could compute the collocations and concordance indexes for a collection of tokens and maintain control of the results.

Example 4-13. Using NLTK to compute collocations in a similar manner to the nltk.Text.collocations demo functionality

```
import json
import nltk

# Load in human readable text from wherever you've saved it

DATA = '10703731246200681024.json'
data = json.loads(open(DATA).read())

# Number of collocations to find

N = 25

all_tokens = [token for activity in data for token in activity['object']['content'
              ].lower().split()]

finder = nltk.BigramCollocationFinder.from_words(all_tokens)
finder.apply_freq_filter(2)
finder.apply_word_filter(lambda w: w in nltk.corpus.stopwords.words('english'))
scorer = nltk.metrics.BigramAssocMeasures.jaccard
collocations = finder.nbest(scorer, N)

for collocation in collocations:
    c = ' '.join(collocation)
    print c
```

In short, the implementation loosely follows NLTK's `collocations` demo function. It filters out bigrams that don't appear more than a minimum number of times (two, in this case) and then applies a scoring metric to rank the results. In this instance,

the scoring function is the well-known Jaccard Index, as defined by `nltk.metrics.Bigram AssocMeasures.jaccard`. A contingency table is used by the `BigramAssocMeasures` class to rank the co-occurrence of terms in any given bigram as compared to the possibilities of other words that could have appeared in the bigram. Conceptually, the Jaccard Index measures similarity of sets, and in this case, the sample sets are specific comparisons of bigrams that appeared in the text. The details of how contingency tables and Jaccard values are calculated is arguably an advanced topic, but the next section, "Contingency Tables and Scoring Functions" on page 158, provides an extended discussion of those details since they're foundational to a deeper understanding of collocation detection.

In the meantime, though, let's examine Example 4-14. This example shows some output from Tim O'Reilly's Google+ data that should make it pretty apparent that returning scored bigrams is immensely more powerful than only returning tokens, because of the additional context that grounds the terms in meaning.

Example 4-14. Sample results from Example 4-13

```
ada lovelace
jennifer pahlka
hod lipson
pine nuts
safe, welcoming
1st floor,
5 southampton
7ha cost:
bcs, 1st
borrow 42
broadcom masters
building, 5
date: friday
disaster relief
dissolvable sugar
do-it-yourself festival,
dot com
fabric samples
finance protection
london, wc2e
maximizing shareholder
patron profiles
portable disaster
rural co
vat tickets:
```

Keeping in mind that no special heuristics or tactics that could have inspected the text for proper names based on Title Case were employed, it's actually quite amazing that so many proper names and common phrases were sifted out of the data. There's still a certain amount of inevitable noise in the results because we have not yet made any effort to clean punctuation from the tokens, but for the small amount of work we've put in,

the results are really quite good. This might be a good time to mention that even if reasonably good natural language processing capabilities were employed, it might still be difficult to eliminate all the noise from the results of textual analysis. Getting comfortable with the noise and finding heuristics to control it is a good idea until you get to the point where you're willing to make a significant investment in obtaining the perfect results that a well-educated human would be able to pick out from the text.

Hopefully, the primary observation you're making at this point is that with very little effort and time invested, we've been able to use another very basic technique to draw out some pretty powerful meaning from some free text data, and the results seem to be pretty representative of what we already suspect should be true. This is encouraging, because it suggests that applying the same technique to anyone else's Google+ data (or any other kind of unstructured text, for that matter) would potentially be just as informative, giving you a quick glimpse into key items that are being discussed. And just as importantly, while the data in this case probably confirms a few things you may already know about Tim O'Reilly, you may have learned a couple of new things, too—one of which is that he might just have a sweet spot for Ada Lovelace as evidenced by "ada lovelace" showing up in the collocation results. While it would be easy enough to use the concordance, a regular expression, or even the Python string type's built-in `find` method to find posts relevant to "ada lovelace", let's instead take advantage of the code we developed in Example 4-9 and use TF-IDF to query for "ada lovelace." What comes back? Survey says:

```
I just got an email from +Suw Charman about Ada Lovelace Day,
and thought I'd share it here, sinc...
    Link: https://plus.google.com/107033731246200681024/posts/1XSAkDs9b44
    Score: 0.198150014715
```

And there you have it. The "ada lovelace" query leads us to some content about Ada Lovelace Day. You've effectively started with a nominal (if that) understanding of the text, narrowed in on some interesting topics using collection analysis, and searched the text for one of those interesting topics using TF-IDF. There's no reason you couldn't also use cosine similarity at this point to find the most similar post to the one about the lovely Ada Lovelace (or whatever it is that you're keen to investigate).

Contingency Tables and Scoring Functions

This section dives into some of the more technical details of how `BigramCollocationFinder`—the Jaccard scoring function from Example 4-13—works. If this is your first reading of the chapter or you're not interested in these details, feel free to skip this section and come back to it later. It's arguably an advanced topic, and you don't need to fully understand it to effectively employ the techniques from this chapter.

A common data structure that's used to compute metrics related to bigrams is the *contingency table*. The purpose of a contingency table is to compactly express the frequencies associated with the various possibilities for appearance of different terms of a bigram. Take a look at the bold entries in Table 4-5, where token1 expresses the existence of token1 in the bigram, and ~token1 expresses that token1 does not exist in the bigram.

Table 4-5. Contingency table example—values in italics represent "marginals," and values in bold represent frequency counts of bigram variations

	token1	~token1	
token2	**frequency(token1, token2)**	**frequency(~token1, token2)**	frequency(*, token2)
~token2	**frequency(token1, ~token2)**	**frequency(~token1, ~token2)**	
	frequency(token1, *)		frequency(*, *)

Although there are a few details associated with which cells are significant for which calculations, hopefully it's not difficult to see that the four middle cells in the table express the frequencies associated with the appearance of various tokens in the bigram. The values in these cells can compute different similarity metrics that can be used to score and rank bigrams in order of likely significance, as was the case with the previously introduced Jaccard Index, which we'll dissect in just a moment. First, however, let's briefly discuss how the terms for the contingency table are computed.

The way that the various entries in the contingency table are computed is directly tied to which data structures you have precomputed or otherwise have available. If you assume that you only have available a frequency distribution for the various bigrams in the text, the way to calculate frequency(token1, token2) is a direct lookup, but what about frequency(~token1, token2)? With no other information available, you'd need to scan *every single bigram* for the appearance of token2 in the second slot and subtract frequency(token1, token2) from that value. (Take a moment to convince yourself that this is true if it isn't obvious.)

However, if you assume that you have a frequency distribution available that counts the occurrences of each individual token in the text (the text's unigrams) in addition to a frequency distribution of the bigrams, there's a much less expensive shortcut you can take that involves two lookups and an arithmetic operation. Subtract the number of times that token2 appeared as a unigram from the number of times the bigram (token1, token2) appeared, and you're left with the number of times the bigram (~token1, token2) appeared. For example, if the bigram ("mr.", "green") appeared three times and the unigram ("green") appeared seven times, it must be the case that the bigram (~"mr.", "green") appeared four times (where ~"mr." literally means "any token other than 'mr.'"). In Table 4-5, the expression frequency(*, token2) represents the unigram token2 and is referred to as a marginal because it's noted in the margin of the table as a shortcut. The value for frequency(token1, *) works the same way in helping to compute frequen-

cy(token1, ~token2), and the expression frequency(*, *) refers to any possible unigram and is equivalent to the total number of tokens in the text. Given frequency(token1, token2), frequency(token1, ~token2), and frequency(~token1, token2), the value of frequency(*, *) is necessary to calculate frequency(~token1, ~token2).

Although this discussion of contingency tables may seem somewhat tangential, it's an important foundation for understanding different scoring functions. For example, consider the Jaccard Index. Conceptually, it expresses the similarity of two sets and is defined by:

$$\frac{|Set1 \cap Set2|}{|Set1 \cup Set2|}$$

In other words, that's the number of items in common between the two sets divided by the total number of distinct items in the combined sets. It's worth taking a moment to ponder this simple yet effective calculation. If *Set1* and *Set2* were identical, the union and the intersection of the two sets would be equivalent to one another, resulting in a ratio of 1.0. If both sets were completely different, the numerator of the ratio would be 0, resulting in a value of 0.0. Then there's everything else in between. The Jaccard Index as applied to a particular bigram expresses the ratio between the frequency of a particular bigram and the sum of the frequencies with which any bigram containing a term in the bigram of interest appears. One interpretation of that metric might be that the higher the ratio is, the more likely it is that (token1, token2) appears in the text, and hence, the more likely it is that the collocation "token1 token2" expresses a meaningful concept.

The selection of the most appropriate scoring function is usually determined based upon knowledge about the characteristics of the underlying data, some intuition, and sometimes a bit of luck. Most of the association metrics defined in `nltk.metrics.associations` are discussed in Chapter 5 of Christopher Manning and Hinrich Schuetze's Foundations of Statistical Natural Language Processing (MIT Press), which is conveniently available online and serves as a useful reference for the descriptions that follow. An in-depth discussion of these metrics is outside the scope of this book, but the promotional chapter just mentioned provides a detailed account with in-depth examples. The Jaccard Index, Dice's coefficient, and the likelihood ratio are good starting points if you find yourself needing to build your own collocation detector. They are described, along with some other key terms, in the list that follows:

Raw frequency

As its name implies, raw frequency is the ratio expressing the frequency of a particular *n*-gram divided by the frequency of all *n*-grams. It is useful for examining the overall frequency of a particular collocation in a text.

Jaccard Index

The Jaccard Index is a ratio that measures the similarity between sets. As applied to collocations, it is defined as the frequency of a particular collocation divided by the total number of collocations that contain at least one term in the collocation of interest. It is useful for determining the likelihood of whether the given terms actually form a collocation, as well as ranking the likelihood of probable collocations. Using notation consistent with previous explanations, this formulation would be mathematically defined as:

$$\frac{\text{frequency(token1, token2)}}{\text{frequency(token1, token2)} + \text{frequency(~token1, token2)} + \text{frequency(token1, ~token2)}}$$

Dice's coefficient

Dice's coefficient is extremely similar to the Jaccard Index. The fundamental difference is that it weights agreements among the sets twice as heavily as Jaccard. It is defined mathematically as:

$$\frac{\text{2*frequency(token1, token2)}}{\text{frequency(*, token2)} + \text{frequency(token1, *)}}$$

Mathematically, it can be shown fairly easily that:

$$\text{Dice} = \frac{2 * \text{Jaccard}}{1 + \text{Jaccard}}$$

You might choose to use this metric instead of the Jaccard Index when it's more important to bias the score in favor of instances in which the collocation "token1 token2" appears.

Likelihood ratio

This metric is yet another approach to hypothesis testing that is used to measure the independence between terms that may form a collocation. It's been shown to be a more appropriate approach for collocation discovery than the chi-square test in the general case, and it works well on data that includes many infrequent collocations. The particular calculations involved in computing likelihood estimates for collocations as implemented by NLTK assume a binomial distribution, where the parameters governing the distribution are calculated based upon the number of occurrences of collocations and constituent terms.

Chi-square

Like Student's t-score, this metric is commonly used for testing independence between two variables and can be used to measure whether two tokens are collocations based upon Pearson's chi-square test of statistical significance. Generally speaking,

the differences obtained from applying the t-test and chi-square test are not substantial. The advantage of chi-square testing is that unlike t-testing, it does not assume an underlying normal distribution; for this reason, chi-square testing is more commonly used.

Student's t-score
: Traditionally, Student's t-score has been used for hypothesis testing, and as applied to *n*-gram analysis, t-scores can be used for testing the hypothesis of whether two terms are collocations. The statistical procedure for this calculation uses a standard distribution per the norm for t-testing. An advantage of the t-score values as opposed to raw frequencies is that a t-score takes into account the frequency of a bigram relative to its constituent components. This characteristic facilitates ranking the strengths of collocations. A criticism of the t-test is that it necessarily assumes that the underlying probability distribution for collocations is normal, which is not often the case.

Pointwise Mutual Information
: Pointwise Mutual Information (PMI) is a measure of how much information is gained about a particular word if you also know the value of a neighboring word. To put it another way, how much one word can tell you about another. Ironically (in the context of the current discussion), the calculations involved in computing the PMI lead it to score high-frequency words lower than low-frequency words, which is opposite of the desired effect. Therefore, it is a good measure of independence but not a good measure of dependence (i.e., a less than ideal choice for scoring collocations). It has also been shown that sparse data is a particular stumbling block for PMI scoring, and that other techniques such as the likelihood ratio tend to outperform it.

Closing Remarks

This chapter introduced the Google+ API and how to collect and cleanse human language data as part of an exercise in querying for a person's Google+ activities. We then spent some time learning about a few of the fundamentals of IR theory, TF-IDF, cosine similarity, and collocations, as the means of analyzing the data we collected and eventually worked up to the point that we were considering some of the same problems that any search engine provider has had to consider to build a successful technology product. However, even though this chapter has hopefully given you some good insight into how to extract useful information from unstructured text, it's barely scratched the surface of the most fundamental concepts, both in terms of theory and engineering considerations. Information retrieval is literally a multibillion-dollar industry, so you can only imagine the amount of combined investment that goes into both the theory and implementations that work at scale to power search engines such as Google and Yahoo!. In closing out this chapter, this section is a modest attempt to make sure you're aware of some of the

inherent limitations of TF-IDF, cosine similarity, and other concepts introduced in this chapter, with the hopes that it will be beneficial in shaping your overall view of this space.

- While TF-IDF is a powerful tool that's easy to use, our specific implementation of it has a few important limitations that we've conveniently overlooked but that you should consider. One of the most fundamental is that it treats a document as a bag of words, which means that the order of terms in both the document and the query itself does not matter. For example, querying for "Green Mr." would return the same results as "Mr. Green" if we didn't implement logic to take the query term order into account or interpret the query as a phrase as opposed to a pair of independent terms. But obviously, the order in which terms appear is very important.
- Even if you carry out an *n*-gram analysis to account for collocations and term ordering, there's still the underlying issue that TF-IDF assumes that all tokens with the same text value mean the same thing. Clearly, however, this need not be the case. Any homonym of your choice is a counterexample, and there are plenty of them, and even words that do mean the same thing can connote slightly different meanings depending on the exact context in which they are used. A key difference in a traditional keyword search technology based on TF-IDF principles and a more advanced semantic search engine is that the semantic search engine would necessarily allow you to ground your search terms in a particular meaning by defining context. For example, you might be able to specify that the term you are searching for should be interpreted as a person, location, organization, or other specific type of entity. Being able to ground search terms in specific contexts is a very active area of research at the moment.
- Cosine similarity suffers from many of the same flaws as TF-IDF. It does not take into account the context of the document or the term order from the *n*-gram analysis, and it assumes that terms appearing close to one another in vector space are necessarily similar, which is certainly not always the case. The obvious counterexample is homonyms, which may mean quite different things but are interpreted as the same term since they have the same text values. Our particular implementation of cosine similarity also hinges on TF-IDF scoring as its means of computing the relative importance of words in documents, so the TF-IDF errors have a cascading effect.
- You've probably also realized that there can be a lot of pesky details that have to be managed in analyzing unstructured text, and these details turn out to be pretty important for state-of-the-art implementations. For example, string comparisons are case-sensitive, so it's important to normalize terms so that frequencies can be calculated as accurately as possible. But blindly normalizing to lowercase, for example, can also potentially complicate the situation, since the case used in certain words and phrases can be important. "Mr. Green" and "Web 2.0" are two examples worth considering. In the case of "Mr. Green", maintaining the title case in "Green"

could potentially be advantageous since it could provide a useful clue to a query algorithm that it's not referring to an adjective and is likely part of a noun phrase. We'll briefly touch on this topic again in Chapter 5 when NLP is discussed, since it's ultimately the context in which "Green" is being used that is lost with the bag-of-words approach, whereas more advanced parsing with NLP has the potential to preserve that context.

- Another consideration that's rooted more in our particular implementation than a general characteristic of TF-IDF itself is that our use of `split` to tokenize the text may leave trailing punctuation on tokens that can affect tabulating frequencies. For example, in Example 4-7, `corpus['b']` ends with the token "study", which is not the same as the token "study" that appears in `corpus['a']` (the token that someone would probably be more likely to query). In this instance, the trailing period on the token affects both the TF and the IDF calculations. As previously suggested, you might consider stemming words so that common variations of the same word are essentially treated as the same term instead of different terms. Check out the `nltk.stem` package for several good stemming implementations.
- Finally, there are plenty of engineering considerations to ponder should you decide to implement a solution that you plan to take into a production situation. The use of indexes and caching are critical considerations for obtaining reasonable query times on even moderately large data sets.

Given the immense power of search providers like Google, it's easy to forget that these foundational search techniques even exist. However, understanding them yields insight into the assumptions and limitations of the commonly accepted status quo for search, while also clearly differentiating the state-of-the-art entity-centric techniques that are emerging; Chapter 5 introduces a fundamental paradigm shift away from some of the techniques in this chapter. There are lots of exciting opportunities for technology-driven companies that can effectively analyze human language data.

Recommended Exercises

- Take advantage of IPython Notebook's plotting features that were introduced in Chapter 1 to graph Zipf's curve for the tokens from a corpus.
- Mine the comments feed and try to identify trends based on frequency of commenting. For example, who are the most frequent commenters across a few hundred activities for a popular Google+ user like Tim O'Reilly.
- Mine Google+ activities to discover which activities are the most popular. A suitable starting point might be the number of comments and number of times a post is reshared.
- Fetch the content from links that are referenced in Google+ activities and adapt the `cleanHtml` function from this chapter to extract the text across the web pages in a

user's activity stream for analysis. Are there any common themes in links that are shared? What are the most frequent words that appear in the text?

- If you'd like to try applying the techniques from this chapter to the Web (in general), you might want to check out Scrapy, an easy-to-use and mature web scraping and crawling framework that can help you to harvest web pages.
- Spend some time and add interactive capabilities to the matrix diagram presented in this chapter. Can you add event handlers to automatically take you to the post when text is clicked on? Can you conceive of any meaningful ways to order the rows and columns so that it's easier to identify patterns?
- Update the code that emits the JSON that drives matrix diagram so that it computes similarity differently, and thus, correlates documents differently from the default implementation.
- What additional features from the text can you think of that would make computing similarity across documents more accurate?
- Spend some time really dig into the theory presented in this chapter for the underlying IR concepts that were presented.

CHAPTER 5
Mining Web Pages: Using Natural Language Processing to Understand Human Language, Summarize Blog Posts, and More

This chapter follows very closely on the heels of the chapter before it and is a modest attempt to introduce Natural Language Processing (NLP) and apply it to the vast source of human language[1] data that you'll encounter on the social web (or elsewhere.) Whereas the previous chapter introduced some foundational techniques from Information Retrieval (IR) theory, which generally treats text as document-centric "bags of words" that can be modeled and manipulated as vectors, this chapter delves much deeper into the realm involving the semantics of human language data such as the words that you are reading on this page, that you'll find in Facebook posts, in web pages linked into tweets, or just about anywhere else. After all, we're talking about making sense of the most ubiquitous kind of data available to us.

In the spirit of the prior chapters, we'll attempt to cover the minimal level of detail required to empower you with a solid general understanding of an inherently complex topic, while also providing enough of a technical drill-down that you'll be able to immediately get to work mining some data. Although we've been regularly cutting corners and attempting to give you the crucial 20% of the skills that you can use to do 80% of the work, and no chapter out of any book (or small multivolume set of books, for that matter) could possibly do the topic of NLP justice, the content in this chapter is a prag-

1. Thoughout this chapter, the phrase "human language data" will commonly be used to refer to the object of natural language processing and is intended to convey the very same meaning as "natural language data" or "unstructured data". No particular distinction is intended to be drawn by this particular choice of words other than its precision in describing the data itself.

matic introduction that'll give you enough information to do some pretty amazing things with the human language data that you'll find all over the social web. Although we'll be focused on extracting human language data from web pages and feeds, keep in mind that just about every social web property with an API is going to return human language, so these techniques generalize to just about any social web property.

Although it's not absolutely necessary that you have read the previous chapter before you dive into this chapter, it's highly recommended that you do so. A good understanding of NLP presupposes an appreciation and working knowledge of some of the fundamental strengths and weaknesses of TF-IDF, vector space models, etc. In that regard, these two chapters have a somewhat tighter coupling than most other chapters in this book.

Scraping, Parsing, and Crawling the Web

Although it's trivial to use a programming language or terminal utility such as `curl` or `wget` to fetch an arbitrary web page, extracting the isolated text that you want from the page isn't quite as trivial. Although the text is certainly in the page, so is lots of other boilerplate content such as navigation bars, headers, footers, advertisements, and other sources of noise that you probably don't care about. As such, the bad news is that the problem isn't quite as simple as just stripping out the HTML tags and processing what is left behind, because lots of noise would likely be left behind. In some cases, there may actually be more text in the page that represents noise than the signal you were looking for all depending on the nature of the website (if you can believe that.) The good news is that the maturity of tools for helping to identify the body of the content that you're interested has continued to grow over the years, and there are some excellent options that do a very reasonable job of isolating the relevant content that you'd want for text-mining purposes. Additionally, the relative ubiquitousness of feeds such as RSS and Atom can often aid the process of retrieving clean text without all of the cruft that's typically in web pages if you have the foresight to fetch the feeds while they are available. It's often the case that feeds are only published for "recent" content, however, so you may often have to process web pages as a default option. If given the choice, you'll probably want to prefer the feeds over arbitrary web pages, but you'll need to be prepared to process both.

The process of extracting the text from a web page is typically referred to as "web scraping," and an excellent tool for web scraping is the Java-based boilerpipe library that is designed to identify and remove the boilerplate from web pages. The boilerplate library is based on a published paper entitled "Boilerplate Detection using Shallow Text Features" by the author, which explains the efficacy of the approach that uses supervised machine learning techniques to bifurcate the boilerplate and the content of the page.

Supervised machine learning techniques involve a process that involves being provided training samples that are learned from in order to create a predictive model that can be used to evaluate arbitrary samples, and as such, boilerpipe is customizable should you desire to tweak it for increased precision.

Even though the library is Java-based, it's sufficiently useful and popular enough that a motivated Python developer has written a package wrapper called python-boilerplate for boilerpipe, so it can be used as a standard Python package, although installation requires just a slight deviation from the standard procedures since it is not yet in the PyPI. Thus, as opposed to using `pip install python-boilerplate`, you'll need to install it (and one of its dependencies called jpype that is also not yet in PyPI) directly from the GitHub repository with the following two commands:

- `pip install git+git://github.com/ptwobrussell/jpype.git#jpype-github`
- `pip install git+git://github.com/ptwobrussell/python-boilerpipe.git#boilerpipe-github`

Assuming you have a relatively recent version of Java on your system, that's all that should be required to use boilerpate. Example 5-1 demonstrates a sample script that illustrates its rather straightforward usage for the task of extracting the body content for an article as denoted by the `ArticleExtractor` parameter that's passed into the `Extractor` constructor. You can also try out a hosted version of boilerpipe online to try out the difference between some of its other provided extractors such as the `LargestContentExtractor` or `DefaultExtractor`. Basically, each of these extractors work exactly as described. There's a default extractor that works for the general case, an extractor that has been trained for web pages containing articles, and an extractor that is trained to extract the largest body of text on a page, which might be suitable for web pages that tend to have just one large block of text. There may be some light post-processing that's still required on the text, depending on what other features you can identify that may be noise or require attention, but the gist of employing boilerpipe to do the heavy-lifting is just about as easy as it should be.

Example 5-1. Using boilerpipe to extract the text from a web page

```
from boilerpipe.extract import Extractor

URL='http://radar.oreilly.com/2010/07/louvre-industrial-age-henry-ford.html'

extractor = Extractor(extractor='ArticleExtractor', url=URL)

print extractor.getText()
```

Although web scraping used to the the only way to fetch content from websites, there's potentially an eaiser way to harvest content, especially if it's content that's coming from a news source, blog, or other syndicated source. But first let's take a quick trip down

memory lane: if you've been using the Web long enough, you may remember a time in the late 1990s when news readers didn't exist. Back then, if you wanted to know what the latest changes to a website were, you just had to go to the site and see if anything had changed, and as a result syndiation formats took advantage of the self-publishing movement with blogs and formats such as RSS (Really Simple Syndication) and Atom built upon the evolving XML specifications that were growing in popularity to handle the general problem of content providers publishing content and consumers subscribing to content. Parsing feeds is an easier problem to solve since the feeds are (usually) well-formed XML data that validates to a published schema whereas web pages may or may not be well-formed, valid or even follow best practices. A commonly used Python package for processing feeds is appropriately named `feedparser` and is an essential utility to have on hand for parsing feeds. It can be installed with `pip` using the standard `pip install feedparser` approach in a terminal, and Example 5-2 illustrates minimal usage to extract the text, title, and source URL for an entry in an RSS feed.

HTML, XML, and XHTML

As the early Web evolved out of its infancy, the problem of separating the content in a page its presentation quickly became recognized as an early problem, and XML was (in part) a solution to the problem. The idea was that content creators could publish data in an XML format and use a stylesheet to transform it into XHTML for presentation to an end user where XHTML is essentially just HTML that is well-formed XML where each tag is defined in lowercase, properly nested as a tree-structure, self-closing such as `<br />` or where each opening tag such as `<p>` has a corresponding closing tag such as `</p>`. In the context of web scraping, these guarantees also have the added benefit of making each web page much easier to process with a parser, and in terms of design, it appeared that XHTML was exactly what the Web needed. There was a lot to gain and virtually nothing to lose from the proposition: *well-formed* XHTML content could be proven *valid* against an XML schema and enjoy all of the other perks of XML, such as custom attributes using namespaces (a device that semantic web technologies such as RDFa rely upon).

The problem is that it just didn't catch on. Whether it was the fault of Internet Explorer, confusion amongst web developers about delivering the correct MIME types to browsers, the quality of the XML developer tools that were available, or the fact that it just wasn't reasonable to expect the entire Web to take a brief timeout to perform the conversion is a contentious discussion that we should probably avoid. The reality is that it just didn't happen as we might have expected. As a result, we now live in a world where semantic markup based on the HTML 4.01 standard that's over a decade old continues to thrive while XHTML and XHTML-based technologies such as RDFa remain on the fringe. (In fact, libraries such as BeautifulSoup are designed with the specific intent of being able to reasonably process HTML that probably isn't well-formed or sane.) Most of the web development world is holding its breath and hoping that HTML5 will indeed create a long-overdue convergence as technologies like microdata catch on and pub-

lishing tools modernize. The HTML article on Wikipedia is an worth reading if you find this kind of history interesting.

Example 5-2. Using feedparser to extract the text (and other fields) from an RSS or Atom feed

```
import feedparser

URL='http://feeds.feedburner.com/oreilly/radar/atom'

fp = feedparser.parse(FEED_URL)

for e in fp.entries:
    print e.title
    print e.links[0].href
    print e.content[0].value
```

Crawling websites is a logical extension of the same concepts already presented in this section and typically consists of fetching a page, extracting the hyperlinks in the page and then systematically fetching all of those pages that are hyperlinked, and you'll repeat the process for some depth, all depending on your objective. This very process is the way that the earliest search engines used to work and most search engines that index the web still continue to work today. Although a crawl of the web is far outside our scope, it is helpful to have a working knowledge of the complexity of the problem, so let's briefly think about the computational complexity of harvesting all of those pages. Nowadays, you can get a periodically updated crawl of the web that' suitable for most research purposes from a source such as Amazon's Common Crawl Corpus, which features more than 5 billion web pages and checks in at over 81 terabytes of data!

In terms of implementing your own web crawl, should you ever desire to do so, Scrapy is a Python-based framework for web scraping and is an excellent resource if your endeavors ential web crawling. The actual exercise of performing a web crawl is just outside the scope of this chapter, but the Scrapy documentation online is excellent and includes all of the tutelage you'll need to get a web crawl going with very little effort up front. The next section is a brief aside that discusses the computational complexity of how web crawls are typically implemented so that you can better appreciate what you may be getting yourself into.

Breadth-first Search

This section contains some detailed content and analysis about how web crawls can be implemented and is not essential to your understanding of the content in this chapter (although you will likely find it interesting and edifying.) If this is your first reading of the chapter, feel free to save it for next time.

The basic algorithm for a web crawl can be framed as a breadth-first search, which is a fundamental technique for exploring a space that's typically modeled as a tree or a graph given a starting node and no other known information except a set of possibilities. In our web crawl scenario, our starting node would be the initial web page and the set of neighboring nodes would be the other pages that are hyperlinked. From an algorithmic standpoint, a breadth-first search provides a deterministic way of exploring the search space in a way that expands broadly. There are alternative ways to search the space with a depth-first search being a common alternative to a breadth-first search. The particular choice of one technique versus another can depend on available computing resources, specific domain knowledge, or even theoretical considerations. At any rate, a breadth-first search is a reasonable approach for exploring a sliver of the Web and Example 5-3 presents some psuedo-code that illustrates how it works and Figure 5-1 provides some visual intuition behind how the search would look if you were to draw it out on the back of a napkin.

Example 5-3. Pseudocode for a breadth-first search

```
Create an empty graph
Create an empty queue to keep track of nodes that need to be processed

Add the starting point to the graph as the root node
Add the root node to a queue for processing

Repeat until some maximum depth is reached or the queue is empty:
  Remove a node from the queue
  For each of the node's neighbors:
    If the neighbor hasn't already been processed:
      Add it to the queue
      Add it to the graph
      Create an edge in the graph that connects the node and its neighbor
```

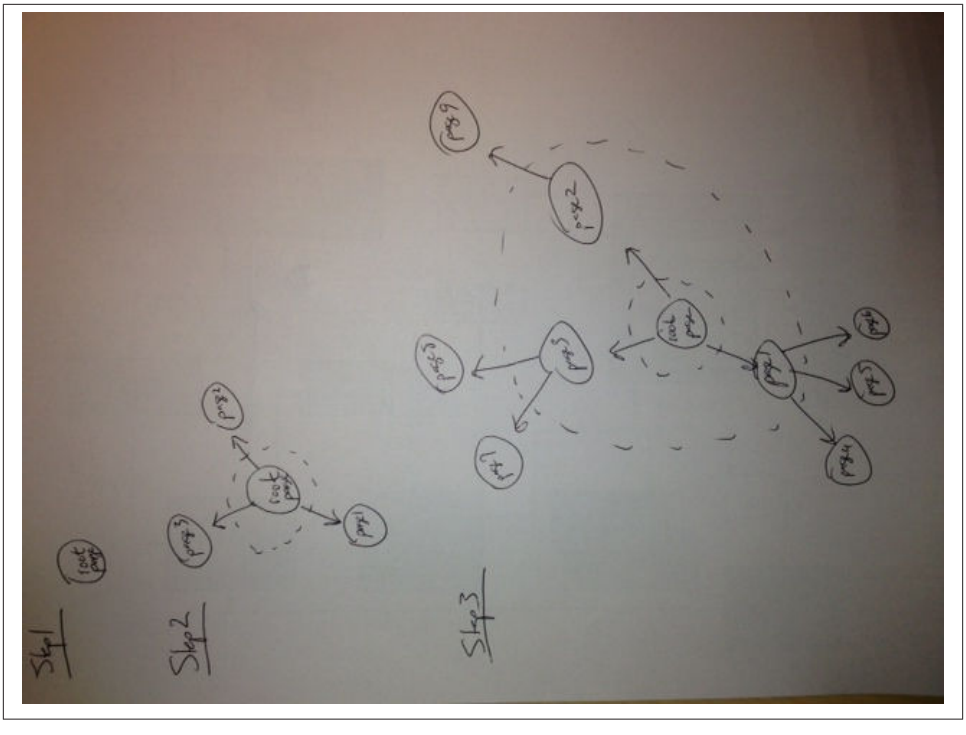


Figure 5-1. Intuition for understanding breadth-first search. Each step of the search expands the depth by one level until a maximum depth or some other termination criteria is reached [XXX: Production - please help with this illustration]

We generally haven't taken quite this long of a pause to analyze the approach, but breadth-first search is a fundamental tool you'll want to have in your belt and is worthwhile to examine more closely. In general, there are two criteria for examination that you should always consider for an algorithm: efficiency and effectiveness. Or, to put it another way: performance and quality.

Standard performance analysis of any algorithm generally involves examining its worst-case time and space complexity—in other words, the amount of time it would take the program to execute, and the amount of memory required for execution over a very large data set. The breadth-first approach we've taken is essentially a breadth-first search, except that we're not actually searching for anything in particular because there are no exit criteria beyond expanding the graph out either to a maximum depth or until we run out of nodes. If we were searching for something specific instead of just crawling links indefinitely, that would be considered an actual breadth-first search. In practice, however, almost all searches impose some kind of exit criteria because of the limitations of finite computing resources. Thus, a more common variation of a breadth-first search

is called a *bounded breadth-first* search, which imposes a limit on the maximum depth of the search just as we do in this example.

For a breadth-first search (or breadth-first crawl), both the time and space complexity can be bounded in the worst case by b^d, where b is the branching factor of the graph and d is the depth. If you sketch out an example on paper and think about it, this analysis makes a lot of sense. If every node in a graph had five neighbors, and you only went out to a depth of one, you'd end up with six nodes in all: the root node and its five neighbors. If all five of those neighbors had five neighbors too, and you expanded out another level, you'd end up with 31 nodes in all: the root node, the root node's five neighbors, and five neighbors for each of the root node's neighbors. Such analysis may seem pedantic, but in the emerging era of *big data*, being able to perform at least a crude level of analysis based on what you know of the data set is more important than ever. (to come) provides an overview of how b^d grows for a few sizes of b and d.

Table 5-1. Example branching factor calculations for graphs of varying depths

Branching factor	Nodes for depth = 1	Nodes for depth = 2	Nodes for depth = 3	Nodes for depth = 4	Nodes for depth = 5
2	3	7	15	31	63
3	4	13	40	121	364
4	5	21	85	341	1365
5	6	31	156	781	3906
6	7	43	259	1555	9331

While the previous comments pertain primarily to the theoretical bounds of the algorithm, one final consideration worth noting is the practical performance of the algorithm for a data set of a fixed size. Mild profiling of a breadth-first implementation that fetches web pages would likely reveal that the code is primarily *I/O bound* from the standpoint that the vast majority of time is spent waiting for a library call in a package such as `urllib2` to return content to be processed. In situations in which you are I/O bound, a thread pool is a common technique for increasing performance.

Decoding Syntax to Discover Semantics

You may recall from the previous chapter that perhaps the most fundamental weaknesses of TF-IDF and cosine similarity are that these models inherently don't require a deep semantic understanding of the data. Quite the contrary, the examples in that chapter were able to take advantage of very basic syntax that separated tokens by whitespace to break an otherwise opaque document into a bag of words and use frequency and simple statistical similarity metrics to determine which tokens were likely to be important in the data. Although you can do some really amazing things with these techniques, they don't really give you any notion of what any given token means in the context in

which it appears in the document. Look no further than a sentence containing a homograph[2] such as "fish", "bear", or even "google" as a case in point; either one could be a noun or a verb.

NLP is inherently complex and difficult to do even reasonably well, and completely nailing it for a large set of commonly spoken languages may very well be the problem of the century. Despite what many mistakenly believe, it's far from an unsolved problem, and it is already the case that we are starting to see an upswing in "deeper understanding" of the web with initiatives such as Google's Knowledge Graph initiatives that are being promoted as "the future of search." After all, a complete mastery of NLP is essentially a plausable strategy for acing the Turing Test, and to the most careful observer, a computer program that achieves this level of "understanding" demonstrates an uncanny amount of human-like intelligence even if it is through a brain that's mathematically modeled in software as opposed to a biological brain with neurons firing and electical impulses traversing the synapses. Whereas structured or semi-structured sources are essentially collections of records with some presupposed meaning given to each field that can immediately be analyzed, there are more subtle considerations to be handled with human language data for even the seemingly simplest of tasks. For example, let's suppose you're given a document and asked to count the number of sentences in it. It's a trivial task if you're a human and have just a basic understanding of English grammar, but it's another story entirely for a machine, which will require a complex and detailed set of instructions to complete the same task.

The encouraging news is that machines can detect the ends of sentences on relatively well-formed data very quickly and with nearly perfect accuracy. However, even if you've accurately detected all of the sentences, there's still a lot that you probably don't know about the ways that words or phrases are used in those sentences. For example, consider the potentially ambiguous phrase, "That car is bad!". It's a trivially parseable sentence consisting of almost nothing except a subject, predicate, and object. However, without additional context, even if you have perfect information about the grammatical structure of the sentence—you have no way of knowing the meaning of the word "bad"—it could be intended to convey that the car is "bad to the bone" but it could also be intended to convey that the car is a jalopy. The point is that even with perfect information about the structure of a sentence, you may still need additional context outside the sentence to properly interpret it. Thus, as an overly broad generalization, we can say that NLP is fundamentally about taking an opaque document that consists of an ordered collection of symbols adhering to proper syntax and a reasonably well-defined grammar, and deducing the associated semantics that are associated with those symbols.

2. A homonym is a special case of a homograph. Two words are homographs if they have the same spelling. Two words are homonyms if they have the same spelling and the same pronunciation. For some reason, "homonym" seems more common in parlance than "homograph" even if it's being misused.

Let's get back to the task of detecting sentences, the first step in most NLP pipelines, to illustrate some of the complexity involved in NLP. It's deceptively easy to overestimate the utility of simple rule-based heuristics, and it's important to work through an exercise so that you realize what some of the key issues are and don't waste time trying to reinvent the wheel.

Your first attempt at solving the sentence detection problem might be to just count the periods, question marks, and exclamation points in the sentence. That's the most obvious heuristic for starting out, but it's quite crude and has the potential for producing an extremely high margin of error. Consider the following (pretty unambiguous) accusation:

> "Mr. Green killed Colonel Mustard in the study with the candlestick. Mr. Green is not a very nice fellow."

Simply tokenizing the sentence by splitting on punctuation (specifically, periods) would produce the following result:

```
>>> txt = "Mr. Green killed Colonel Mustard in the study with the \
... candlestick. Mr. Green is not a very nice fellow."
>>> txt.split(".")
['Mr', 'Green killed Colonel Mustard in the study with the candlestick',
 'Mr', 'Green is not a very nice fellow', '']
```

It should be immediately obvious that performing sentence detection by blindly breaking on periods without incorporating some notion of context or higher-level information is insufficient. In this case, the problem is the use of "Mr.", a valid abbreviation that's commonly used in the English language. Although we already know from the previous chapter that *n*-gram analysis of this sample would likely tell us that "Mr. Green" is really one collocation or chunk (a compound token), if we had a larger amount of text to analyze, it's not hard to imagine other edge cases that would be difficult to detect based on the appearance of collocations. Thinking ahead a bit, it's also worth pointing out that finding the key topics in a sentence isn't easy to accomplish with trivial logic either. As an intelligent human, you can easily discern that the key topics in our sample might be "Mr. Green", "Colonel Mustard", "the study", and "the candlestick", but training a machine to tell you the same things without human intervention is a complex task. Take a moment to think about how you might write a computer program to solve the problem.

No, really. Take a moment to sketch out some ideas before reading.

A few obvious possibilities are probably occurring to you, such as doing some 'Title Case' detection with a regular expression, constructing a list of common abbreviations to parse out the proper noun phrases, and applying some variation of that logic to the problem of finding end-of-sentence boundaries to prevent yourself from getting into trouble on that front. OK, sure. Those things will work for some examples, but what's

the margin of error going to be like for arbitrary English text? How forgiving is your algorithm for poorly formed English text; highly abbreviated information such as text messages or tweets; or (gasp) other romantic languages, such as Spanish, French, or Italian? There are no simple answers here, and that's why text analytics is such an important topic in an age where the amount of accessible textual data is literally increasing every second. These things aren't pointed out to discourage you; they're actually mentioned to motivate you to keep trying when times get tough because this is an inherently difficult space that no one has completely conquered yet. Even if you do feel deflated, you won't feel that way for long because the Natural Language Toolkit (NLTK) actually performs reasonably well out-of-the-box for many situations involving arbitrary text, as we're about to see.

Let's prepare to step through a session in the interpreter and perform NLP in an attempt to understand human language data with NLTK. The NLP pipeline we'll follow is typical and resembles the following high-level flow:

End of Sentence (EOS) Detection→
Tokenization→
Part-of-Speech Tagging→
Chunking→
Extraction

We'll continue to use the following sample text from the previous chapter for purposes of illustration: "Mr. Green killed Colonel Mustard in the study with the candlestick. Mr. Green is not a very nice fellow." Remember that even though you have already read the text and understand that it's composed of two sentences and all sorts of other things, it's merely an opaque string value to a machine at this point. Let's look at the steps we need to work through in more detail:

EOS detection
: This step breaks a text into a collection of meaningful sentences. Since sentences generally represent logical units of thought, they tend to have a predictable syntax that lends itself well to further analysis. Most NLP pipelines you'll see begin with this step because tokenization (the next step) operates on individual sentences. Breaking the text into paragraphs or sections might add value for certain types of analysis, but it is unlikely to aid in the overall task of EOS detection. In the interpreter, you'd parse out a sentence with NLTK like so:

```
>>> import nltk
>>> txt = "Mr. Green killed Colonel Mustard in the study with the candlestick. \
... Mr. Green is not a very nice fellow."
>>> sentences = nltk.tokenize.sent_tokenize(txt)
>>> sentences
['Mr. Green killed Colonel Mustard in the study with the candlestick.',
 'Mr. Green is not a very nice fellow.']
```

We'll talk a little bit more about what is happening under the hood with `sent_tokenize` in the next section. For now, we'll accept at face value that proper sentence detection has occurred for arbitrary text—a clear improvement over breaking on characters that are likely to be punctuation marks.

Tokenization

This step operates on individual sentences, splitting them into tokens. Following along in the example interpreter session, you'd do the following:

```
>>> tokens = [nltk.tokenize.word_tokenize(s) for s in sentences]
>>> tokens
[['Mr.', 'Green', 'killed', 'Colonel', 'Mustard', 'in', 'the', 'study', 'with',
  'the', 'candlestick', '.'],
['Mr.', 'Green', 'is', 'not', 'a', 'very', 'nice', 'fellow', '.']]
```

Note that for this simple example, tokenization appeared to do the same thing as splitting on whitespace, with the exception that it tokenized out end-of-sentence markers (the periods) correctly. As we'll see in a later section, though, it can do a bit more if we give it the opportunity, and we already know that distinguishing between whether a period is an end of sentence marker or part of an abbreviation isn't always trivial. As an anecdotal note in passing, some written languages such as ones that use pictograms as opposed to the notion of letters don't necessarily even require whitespace to separate the tokens in sentences and require the reader (or machine) to distinguish the boundaries (which in turn would require a machine model to do the same.)

POS tagging

This step assigns part-of-speech information to each token. In the example interpreter session, you'd run the tokens through one more step to have them decorated with tags:

```
>>> pos_tagged_tokens = [nltk.pos_tag(t) for t in tokens]
>>> pos_tagged_tokens
[[('Mr.', 'NNP'), ('Green', 'NNP'), ('killed', 'VBD'), ('Colonel', 'NNP'),
  ('Mustard', 'NNP'), ('in', 'IN'), ('the', 'DT'), ('study', 'NN'),
  ('with', 'IN'), ('the', 'DT'), ('candlestick', 'NN'), ('.', '.')],
 [('Mr.', 'NNP'), ('Green', 'NNP'), ('is', 'VBZ'), ('not', 'RB'), ('a', 'DT'),
  ('very', 'RB'), ('nice', 'JJ'), ('fellow', 'JJ'), ('.', '.')]]
```

You may not intuitively understand all of these tags, but they do represent part-of-speech information. For example, 'NNP' indicates that the token is a noun that is part of a noun phrase, 'VBD' indicates a verb that's in simple past tense, and 'JJ' indicates an adjective. The Penn Treebank Project provides a full summary of the part-of-speech tags that could be returned. With POS tagging completed, it should be getting pretty apparent just how powerful analysis can become. For example, by using the POS tags, we'll be able to chunk together nouns as part of noun phrases and then try to reason about what types of entities they might be (e.g., people, places, organizations, etc.). If you never thought that you'd need to apply those exercises

from elementary grammar school regarding parts of speech, think again: it's essential to a proper application of natural language processing.

Chunking
: This step involves analyzing each tagged token within a sentence and assembling compound tokens that express logical concepts—quite a different approach than statistically analyzing collocations. It is possible to define a custom grammar through NLTK's `chunk.RegexpParser`, but that's beyond the scope of this chapter; see Chapter 9 ("Building Feature Based Grammars") of *Natural Language Processing with Python* (O'Reilly), available online at *http://nltk.googlecode.com/svn/trunk/doc/book/ch09.html*, for full details. Besides, NLTK exposes a function that combines chunking with named entity extraction, which is the next step.

Extraction
: This step involves analyzing each chunk and further tagging the chunks as named entities, such as people, organizations, locations, etc. The continuing saga of NLP in the interpreter demonstrates:

```
>>> ne_chunks = nltk.batch_ne_chunk(pos_tagged_tokens)
>>> ne_chunks
[Tree('S', [Tree('PERSON', [('Mr.', 'NNP')]), Tree('PERSON', [('Green', 'NNP')]),
    ('killed', 'VBD'),
 Tree('ORGANIZATION', [('Colonel', 'NNP'), ('Mustard', 'NNP')]), ('in', 'IN'),
    ('the', 'DT'), ('study', 'NN'), ('with', 'IN'), ('the', 'DT'),
    ('candlestick', 'NN'), ('.', '.')]),
 Tree('S', [Tree('PERSON', [('Mr.', 'NNP')]), Tree('ORGANIZATION',
    [('Green', 'NNP')]), ('is', 'VBZ'), ('not', 'RB'),
 ('a', 'DT'), ('very', 'RB'), ('nice', 'JJ'), ('fellow', 'JJ'), ('.', '.')])]
>>> ne_chunks[0].draw() # You can draw each chunk in the tree
```

: Don't get too wrapped up in trying to decipher exactly what the tree output means just yet. In short, it has chunked together some tokens and attempted to classify them as being certain types of entities. (You may be able to discern that it has identified "Mr. Green" as a person, but unfortunately categorized "Colonel Mustard" as an organization.) Figure 5-2 illustrates the effect of calling `draw()` on the results from `nltk.batch_ne_chunk`.

As interesting as it would be to continue talking about the intricacies of NLP, producing a state-of-the-art NLP stack or even taking much of a deeper dive into NLTK, that level of engagement isn't really our purpose here. The background in this section is provided to motivate an appreciation for the difficulty of the task and to encourage you to review the NLTK book or one of the many other plentiful resources available online if you'd like to pursue the topic further. As a bit of passing advice, if your business (or idea) depends on a *truly state-of-the-art* NLP stack, strongly consider purchasing a license to a turn-key product from a commercial or academic institution instead of trying to home-brew your own. There's a lot you can do on your own with open source software, and that's a great place to start, but as with anything else, the investment involved can

be significant if you have to make numerous improvements that require specialized consulting engagements. NLP is still very much an active field of research and far from a commodity; although many open source implementations work well on clean English text data such as what you might find in Wikipedia articles or professional news articles, your mileage will surely vary on messier human language data or human languages that haven't had the same level of rigor applied as English.

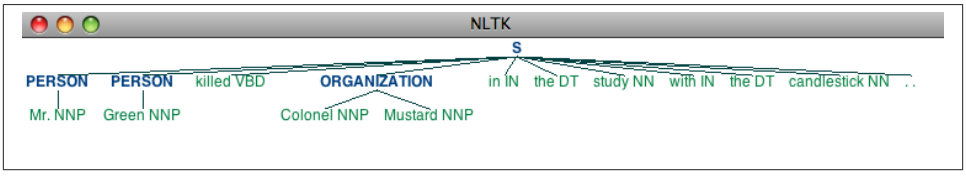


Figure 5-2. NLTK can interface with drawing toolkits so that you can inspect the chunked output in a more intuitive visual form than the raw text output you see in the interpreter

Given that it's possible to customize certain aspects of NLTK, the remainder of this chapter assumes you'll be using NLTK "as-is" unless otherwise noted. (If you had a PhD in computational linguistics or something along those lines, you'd be more than capable of modifying NLTK for your own needs and, frankly, you would probably be reading much more scholarly material than this chapter.)

With that brief introduction to NLP concluded, let's get to work mining some blog data.

Sentence Detection in Human Language Data

Given that sentence detection is probably the first task you'll want to ponder when building an NLP stack, it makes sense to start there. Even if you never complete the remaining tasks in the pipeline, it turns out that EOS detection alone yields some powerful possibilities such as document summarization, which we'll be considering as a follow-up exercise in the next section. But first, we'll need to fetch some clean human language data. Let's use the tried and true `feedparser` package along with some utilities introduced in the previous chapter that are based on `nltk` and `BeautifulSoup` to clean up HTML formatting that may appear in the content to fetch some posts from the O'Reilly Radar blog. The listing in Example 5-4 fetches a few posts and saves them to a local file as JSON.

Example 5-4. Harvesting blog data by parsing feeds

```
import os
import sys
import json
import feedparser
from BeautifulSoup import BeautifulStoneSoup
```

```
from nltk import clean_html

FEED_URL = 'http://feeds.feedburner.com/oreilly/radar/atom'

def cleanHtml(html):
    return BeautifulStoneSoup(clean_html(html),
                              convertEntities=BeautifulStoneSoup.HTML_ENTITIES).contents[0]

fp = feedparser.parse(FEED_URL)

print "Fetched %s entries from '%s'" % (len(fp.entries[0].title), fp.feed.title)

blog_posts = []
for e in fp.entries:
    blog_posts.append({'title': e.title, 'content'
                       : cleanHtml(e.content[0].value), 'link': e.links[0].href})

out_file = '%s.json' % (fp.feed.title,)
f = open(out_file, 'w')
f.write(json.dumps(blog_posts, indent=1))
f.close()

print 'Wrote output file to %s' % (f.name, )
```

Obtaining human language data from a reputable source affords us the luxury of assuming good English grammar; hopefully this also means that one of NLTK's out-of-the-box sentence detectors will work reasonably well. There's no better way to find out than hacking some code to see what happens, so go ahead and take a gander at the code listing in Example 5-5. It introduces the `sent_tokenize` and `word_tokenize` methods, which are aliases for NLTK's currently recommended sentence detector and word tokenizer. A brief discussion of the listing is provided afterward.

Example 5-5. Using NLTK's NLP tools to process human language in blog data)

```
import json
import nltk

BLOG_DATA = "O'Reilly Radar - Insight, analysis, and research about emerging technologies.json"

blog_data = json.loads(open(BLOG_DATA).read())

# Customize your list of stopwords as needed. Here, we add common
# punctuation and contraction artifacts

stop_words = nltk.corpus.stopwords.words('english') + [
    '.',
    ',',
    '--',
    '\'s',
    '?',
    ')',
```

```
    '(',
    ':',
    '\'',
    '\'re',
    '"',
    '-',
    '}',
    '{',
    ]

for post in blog_data:
    sentences = nltk.tokenize.sent_tokenize(post['content'])

    words = [w.lower() for sentence in sentences for w in
             nltk.tokenize.word_tokenize(sentence)]

    fdist = nltk.FreqDist(words)

    # Basic stats

    num_words = sum([i[1] for i in fdist.items()])
    num_unique_words = len(fdist.keys())

    # Hapaxes are words that appear only once

    num_hapaxes = len(fdist.hapaxes())

    top_10_words_sans_stop_words = [w for w in fdist.items() if w[0]
                                    not in stop_words][:10]

    print post['title']
    print '\tNum Sentences:'.ljust(25), len(sentences)
    print '\tNum Words:'.ljust(25), num_words
    print '\tNum Unique Words:'.ljust(25), num_unique_words
    print '\tNum Hapaxes:'.ljust(25), num_hapaxes
    print '\tTop 10 Most Frequent Words (sans stop words):\n\t\t', \
            '\n\t\t'.join(['%s (%s)'
            % (w[0], w[1]) for w in top_10_words_sans_stop_words])
    print
```

The first things you're probably wondering about are the `sent_tokenize` and `word_tokenize` calls. NLTK provides several options for tokenization, but it provides "recommendations" as to the best available via these aliases. At the time of this writing (you can double-check this with `pydoc` or a command like `nltk.tokenize.sent_tokenize?` in IPython or IPython Notebook at any time), the sentence detector is the `PunktSentenceTokenizer` and the word tokenizer is the `TreebankWordTokenizer`. Let's take a brief look at each of these.

Internally, the `PunktSentenceTokenizer` relies heavily on being able to detect abbreviations as part of collocation patterns, and it uses some regular expressions to try to

intelligently parse sentences by taking into account common patterns of punctuation usage. A full explanation of the innards of the `PunktSentenceTokenizer`'s logic is outside the scope of this book, but Tibor Kiss and Jan Strunk's original paper, "Unsupervised Multilingual Sentence Boundary Detection" discusses its approach, is highly readable, and you should take some time to review it. As we'll see in a bit, it is possible to instantiate the `PunktSentenceTokenizer` with sample text that it trains on to try to improve its accuracy. The type of underlying algorithm that's used is an *unsupervised learning algorithm*; it does not require you to explicitly mark up the sample training data in any way. Instead, the algorithm inspects certain features that appear in the text itself, such as the use of capitalization, the co-occurrences of tokens, etc., to derive suitable parameters for breaking the text into sentences.

While NLTK's `WhitespaceTokenizer`, which creates tokens by breaking a piece of text on whitespace, would have been the simplest word tokenizer to introduce, you're already familiar with some of the shortcomings of blindly breaking on whitespace. Instead, NLTK currently recommends the `TreebankWordTokenizer`, a word tokenizer that operates on sentences and uses the same conventions as the Penn Treebank Project.[3] The one thing that may catch you off guard is that the `TreebankWordTokenizer`'s tokenization does some less-than-obvious things, such as separately tagging components in contractions and nouns having possessive forms. For example, the parsing for the sentence "I'm hungry," would yield separate components for "I" and "'m", maintaining a distinction between the subject and verb for "I'm". As you might imagine, finely grained access to this kind of grammatical information can be quite valuable when it's time to do advanced analysis that scrutinizes relationships between subjects and verbs in sentences.

If you have a lot of trouble with advanced word tokenizers such as NLTK's `TreebankWordTokenizer` or `PunktWordTokenizer`, it's fine to default back to the `WhitespaceTokenizer` until you decide whether it's worth the investment to use a more advanced tokenizer. In fact, in some cases using a more straightforward tokenizer can be advantageous. For example, using an advanced tokenizer on data that frequently inlines URLs might be a bad idea, because these tokenizers do not recognize URLs out of the box and will mistakenly break them up into multiple tokens. It's not in the scope of this book to implement a high quality custom tokenizer, which is something that isn't a trivial exercise by any means. Still, there are lots of online sources you can consult if this is something you're interested in attempting.

3. "Treebank" is a very specific term that refers to a corpus that's been specially tagged with advanced linguistic information. In fact, the reason such a corpus is called a "treebank" is to emphasize that it's a bank (think: collection) of sentences that have been parsed into trees adhering to a particular grammar.

Given a sentence tokenizer and a word tokenizer, we can first parse the text into sentences and then parse each sentence into tokens. Note that while this approach is fairly intuitive, it can have a subtle Achilles' heel in that errors produced by the sentence detector propagate forward and can potentially bound the upper limit of the quality that the rest of the NLP stack can produce. For example, if the sentence tokenizer mistakenly breaks a sentence on the period after "Mr." that appears in a section of text such as "Mr. Green killed Colonel Mustard in the study with the candlestick", it may not be possible to extract the entity "Mr. Green" from the text unless specialized repair logic is in place. Again, it all depends on the sophistication of the full NLP stack and how it accounts for error propagation. The out-of-the-box `PunktSentenceTokenizer` is trained on the Penn Treebank corpus and performs quite well. The end goal of the parsing is to instantiate an `nltk.FreqDist` object (which is like a slightly more sophisticated `collections.Counter`), which expects a list of tokens. The remainder of the code in Example 5-5 is straightforward usage of a few of the commonly used NLTK APIs.

The aim of this section was to familiarize you with the first step involved in building an NLP pipeline. Along the way, we developed a few metrics that make a feeble attempt at characterizing some blog data. Our pipeline doesn't involve part-of-speech tagging or chunking (yet), but it should give you a basic understanding of some concepts and get you thinking about some of the subtler issues involved. While it's true that we could have simply split on whitespace, counted terms, tallied the results, and still gained a lot of information from the data, it won't be long before you'll be glad that you took these initial steps toward a deeper understanding of the data. To illustrate one possible application for what you've just learned, in the next section, we'll look at a simple document summarization algorithm that relies on little more than sentence segmentation and frequency analysis.

Document Summarization

Although it may not be immediately obvious, just being able to perform reasonably good sentence detection as part of an NLP approach to mining unstructured data can enable some pretty powerful text-mining capabilities, such as crude but very reasonable attempts at document summarization. There are numerous possibilities and approaches, but one of the simplest to get started with dates all the way back to the April 1958 issue of IBM Journal. In the seminal article entitled "The Automatic Creation of Literature Abstracts," H.P. Luhn describes a technique that essentially boils down to filtering out sentences containing frequently occurring words that appear near one another.

The original paper is easy to understand and is rather interesting; Luhn actually describes how he prepared punch cards in order to run various tests with different parameters! It's amazing to think that what we can implement in a few dozen lines of

Python on a cheap piece of commodity hardware, he probably labored over for hours and hours to program into a gargantuan mainframe. Example 5-6 provides a basic implementation of Luhn's algorithm for document summarization. A brief analysis of the algorithm appears in the next section. Before skipping ahead to that discussion, first take a moment to trace through the code to learn more about how it works.

Example 5-6 uses the `numpy` package (a collection of highly optimized numeric operations), which should have been installed alongside `nltk`. If for some reason you should need to install it, just use `pip install numpy`.

Example 5-6. A document summarization algorithm based principally upon sentence detection and frequency analysis within sentences

```
import json
import nltk
import numpy

BLOG_DATA = "O'Reilly Radar - Insight, analysis, and research about emerging technologies.json"

N = 100  # Number of words to consider
CLUSTER_THRESHOLD = 5  # Distance between words to consider
TOP_SENTENCES = 5  # Number of sentences to return for a "top n" summary

# Approach taken from "The Automatic Creation of Literature Abstracts" by H.P. Luhn

def _score_sentences(sentences, important_words):
    scores = []
    sentence_idx = -1

    for s in [nltk.tokenize.word_tokenize(s) for s in sentences]:

        sentence_idx += 1
        word_idx = []

        # For each word in the word list...
        for w in important_words:
            try:
                # Compute an index for where any important words occur in the sentence

                word_idx.append(s.index(w))
            except ValueError, e: # w not in this particular sentence
                pass

        word_idx.sort()

        # It is possible that some sentences may not contain any important words at all
        if len(word_idx)== 0: continue
```

```
        # Using the word index, compute clusters by using a max distance threshold
        # for any two consecutive words

        clusters = []
        cluster = [word_idx[0]]
        i = 1
        while i < len(word_idx):
            if word_idx[i] - word_idx[i - 1] < CLUSTER_THRESHOLD:
                cluster.append(word_idx[i])
            else:
                clusters.append(cluster[:])
                cluster = [word_idx[i]]
            i += 1
        clusters.append(cluster)

        # Score each cluster. The max score for any given cluster is the score
        # for the sentence

        max_cluster_score = 0
        for c in clusters:
            significant_words_in_cluster = len(c)
            total_words_in_cluster = c[-1] - c[0] + 1
            score = 1.0 * significant_words_in_cluster \
                * significant_words_in_cluster / total_words_in_cluster

            if score > max_cluster_score:
                max_cluster_score = score

        scores.append((sentence_idx, score))

    return scores

def summarize(txt):
    sentences = [s for s in nltk.tokenize.sent_tokenize(txt)]
    normalized_sentences = [s.lower() for s in sentences]

    words = [w.lower() for sentence in normalized_sentences for w in
             nltk.tokenize.word_tokenize(sentence)]

    fdist = nltk.FreqDist(words)

    top_n_words = [w[0] for w in fdist.items()
            if w[0] not in nltk.corpus.stopwords.words('english')][:N]

    scored_sentences = _score_sentences(normalized_sentences, top_n_words)

    # Summarization Approach 1:
    # Filter out non-significant sentences by using the average score plus a
    # fraction of the std dev as a filter

    avg = numpy.mean([s[1] for s in scored_sentences])
```

```
    std = numpy.std([s[1] for s in scored_sentences])
    mean_scored = [(sent_idx, score) for (sent_idx, score) in scored_sentences
                   if score > avg + 0.5 * std]

    # Summarization Approach 2:
    # Another approach would be to return only the top N ranked sentences

    top_n_scored = sorted(scored_sentences, key=lambda s: s[1])[-TOP_SENTENCES:]
    top_n_scored = sorted(top_n_scored, key=lambda s: s[0])

    # Decorate the post object with summaries

    return dict(top_n_summary=[sentences[idx] for (idx, score) in top_n_scored],
                mean_scored_summary=[sentences[idx] for (idx, score) in mean_scored])

blog_data = json.loads(open(BLOG_DATA).read())

for post in blog_data:

    post.update(summarize(post['content']))

    print post['title']
    print '=' * len(post['title'])
    print
    print 'Top N Summary'
    print '-------------'
    print ' '.join(post['top_n_summary'])
    print
    print 'Mean Scored Summary'
    print '-------------------'
    print ' '.join(post['mean_scored_summary'])
    print
```

As example input/output, we'll use Tim O'Reilly's Radar post, "The Louvre of the Industrial Age". It's around 460 words long and is reprinted here so that you can compare the sample output from the two summarization attempts in the listing:

> This morning I had the chance to get a tour of The Henry Ford Museum in Dearborn, MI, along with Dale Dougherty, creator of Make: and Makerfaire, and Marc Greuther, the chief curator of the museum. I had expected a museum dedicated to the auto industry, but it's so much more than that. As I wrote in my first stunned tweet, "it's the Louvre of the Industrial Age."
>
> When we first entered, Marc took us to what he said may be his favorite artifact in the museum, a block of concrete that contains Luther Burbank's shovel, and Thomas Edison's signature and footprints. Luther Burbank was, of course, the great agricultural inventor who created such treasures as the nectarine and the Santa Rosa plum. Ford was a farm boy who became an industrialist; Thomas Edison was his friend and mentor. The museum, opened in 1929, was Ford's personal homage to the transformation of the world that he was so much a part of. This museum chronicles that transformation.

> The machines are astonishing—steam engines and coal-fired electric generators as big as houses, the first lathes capable of making other precision lathes (the makerbot of the 19th century), a ribbon glass machine that is one of five that in the 1970s made virtually all of the incandescent lightbulbs in the world, combine harvesters, railroad locomotives, cars, airplanes, even motels, gas stations, an early McDonalds' restaurant and other epiphenomena of the automobile era.
>
> Under Marc's eye, we also saw the transformation of the machines from purely functional objects to things of beauty. We saw the advances in engineering—the materials, the workmanship, the design, over a hundred years of innovation. Visiting The Henry Ford, as they call it, is a truly humbling experience. I would never in a hundred years have thought of making a visit to Detroit just to visit this museum, but knowing what I know now, I will tell you confidently that it is as worth your while as a visit to Paris just to see the Louvre, to Rome for the Vatican Museum, to Florence for the Uffizi Gallery, to St. Petersburg for the Hermitage, or to Berlin for the Pergamon Museum. This is truly one of the world's great museums, and the world that it chronicles is our own.
>
> I am truly humbled that the Museum has partnered with us to hold Makerfaire Detroit on their grounds. If you are anywhere in reach of Detroit this weekend, I heartily recommend that you plan to spend both days there. You can easily spend a day at Makerfaire, and you could easily spend a day at The Henry Ford. P.S. Here are some of my photos from my visit. (More to come soon. Can't upload many as I'm currently on a plane.)

Filtering sentences using an average score and standard deviation yields a summary of around 170 words:

> This morning I had the chance to get a tour of The Henry Ford Museum in Dearborn, MI, along with Dale Dougherty, creator of Make: and Makerfaire, and Marc Greuther, the chief curator of the museum. I had expected a museum dedicated to the auto industry, but it's so much more than that. As I wrote in my first stunned tweet, "it's the Louvre of the Industrial Age. This museum chronicles that transformation. The machines are astonishing - steam engines and coal fired electric generators as big as houses, the first lathes capable of making other precision lathes (the makerbot of the 19th century), a ribbon glass machine that is one of five that in the 1970s made virtually all of the incandescent lightbulbs in the world, combine harvesters, railroad locomotives, cars, airplanes, even motels, gas stations, an early McDonalds' restaurant and other epiphenomena of the automobile era. You can easily spend a day at Makerfaire, and you could easily spend a day at The Henry Ford.

An alternative summarization approach, which considers only the top N sentences (where $N = 5$ in this case), produces a slightly more abridged result of around 90 words. It's even more succinct, but arguably still a pretty informative distillation:

> This morning I had the chance to get a tour of The Henry Ford Museum in Dearborn, MI, along with Dale Dougherty, creator of Make: and Makerfaire, and Marc Greuther, the chief curator of the museum. I had expected a museum dedicated to the auto industry, but it's so much more than that. As I wrote in my first stunned tweet, "it's the Louvre of

the Industrial Age. This museum chronicles that transformation. You can easily spend a day at Makerfaire, and you could easily spend a day at The Henry Ford.

As in any other situation involving analysis, there's a lot of insight to be gained from visually inspecting the summarizations in relation to the full text. Outputting a simple markup format that can be opened by virtually any web browser is as simple as adjusting the final portion of the script that performs the output to do some string substitution. Example 5-7 illustrates one possibility for visualizing the output of document summarization by presenting the full text of the article with the sentences that are included as part of the summary in bold so that it's easy to see what was included in the summary and what wasn't included in the summary. The script saves a collection of HTML files to disk that you can open in a browser without the help of a web server.

Example 5-7. Visualizing document summarization results

```
import json
import nltk
import numpy

BLOG_DATA = "O'Reilly Radar - Insight, analysis, and research about emerging technologies.json"

HTML_TEMPLATE = """<html>
    <head>
        <title>%s</title>
        <meta http-equiv="Content-Type" content="text/html; charset=UTF-8"/>
    </head>
    <body>%s</body>
</html>"""

blog_data = json.loads(open(BLOG_DATA).read())

for post in blog_data:

    # uses previously defined summarize function
    post.update(summarize(post['content']))

    # You could also store a version of the full post with key sentences markedup
    # for analysis with simple string replacement...

    for summary_type in ['top_n_summary', 'mean_scored_summary']:
        post[summary_type + '_marked_up'] = '<p>%s</p>' % (post['content'], )
        for s in post[summary_type]:
            post[summary_type + '_marked_up'] = \
            post[summary_type + '_marked_up'].replace(s, \
              '<strong>%s</strong>' % (s, ))

        filename = post['title'] + '.summary.' + summary_type + '.html'
        f = open(filename, 'w')
        html = HTML_TEMPLATE % (post['title'] + \
          ' Summary', post[summary_type + '_marked_up'],)
```

```
f.write(html.encode('utf-8'))
f.close()

print "Data written to", f.name
```

The resulting output is the full text of the document with sentences comprising the summary highlighted in bold, as displayed in Figure 5-3. As you explore alternative techniques for summarization, a quick glance between browser tabs can give you an intuitive feel for the similarity between the summarization techniques. The primary difference illustrated here is a fairly long (and descriptive) sentence near the middle of the document, beginning with the words "The machines are astonishing".

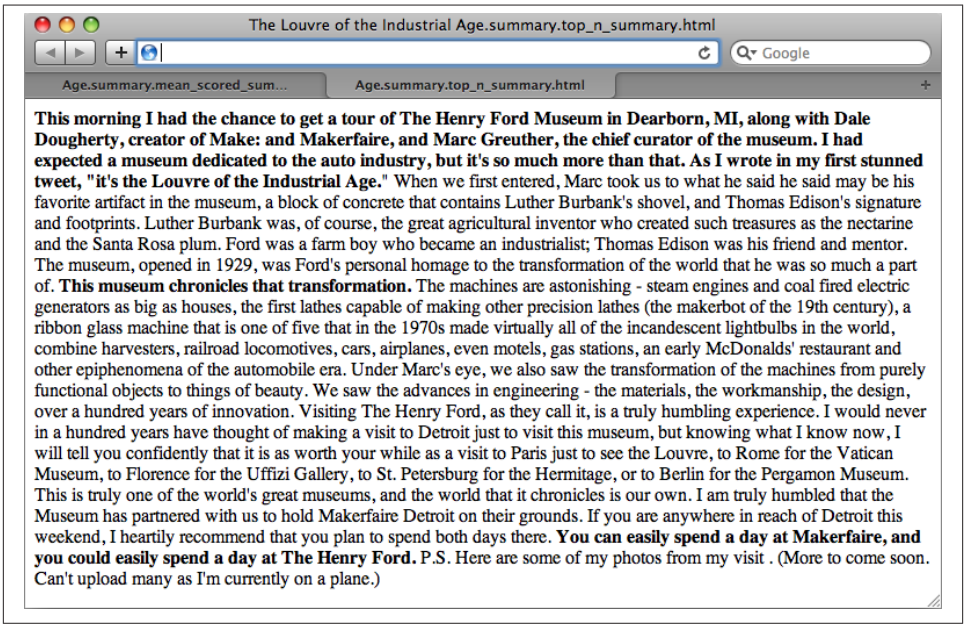


Figure 5-3. The text from an O'Reilly Radar blog post with the most important sentences as determined by a summarization algorithm conveyed in bold

The next section presents a brief discussion of Luhn's document summarization approach.

Analysis of Luhn's Summarization Algorithm

This section provides an analysis of Luhn's summarization algorithm, which is hopefully interesting and edifying, but certainly not a requirement for mining the social web. If you find yourself getting lost in the details, feel free to skip this section and return to it at a later time.

The basic premise behind Luhn's algorithm is that the important sentences in a document will be the ones that contain frequently occurring words. However, there are a few details worth pointing out. First, not all frequently occurring words are important; generally speaking, stopwords are filler and are hardly ever of interest for analysis. Keep in mind that although we do filter out common stopwords in the sample implementation, it may be possible to create a custom list of stopwords for any given blog or domain with additional a priori knowledge, which might further bolster the strength of this algorithm or any other algorithm that assumes stopwords have been filtered. For example, a blog written exclusively about baseball might so commonly use the word "baseball" that you should consider adding it to a stopword list, even though it's not a general-purpose stopword. (As a side note, it would be interesting to incorporate TF-IDF into the scoring function for a particular data source as a means of accounting for common words in the parlance of the domain.)

Assuming that a reasonable attempt to eliminate stopwords has been made, the next step in the algorithm is to choose a reasonable value for N and choose the top N words as the basis of analysis. Note that the latent assumption behind this algorithm is that these top N words are sufficiently descriptive to characterize the nature of the document, and that for any two sentences in the document, the sentence that contains more of these words will be considered more descriptive. All that's left after determining the "important words" in the document is to apply a heuristic to each sentence and filter out some subset of sentences to use as a summarization or abstract of the document. Scoring each sentence takes place in the function `score_sentences`. This is where most of the interesting action happens in the listing.

In order to score each sentence, the algorithm in `score_sentences` applies a simple distance threshold to cluster tokens, and scores each cluster according to the following formula:

$$\frac{(\text{significant words in cluster})^2}{\text{total words in cluster}}$$

The final score for each sentence is equal to the highest score for any cluster appearing in the sentence. Let's consider the high-level steps involved in `score_sentences` for an example sentence to see how this approach works in practice:

Input: Sample sentence

```
['Mr.', 'Green', 'killed', 'Colonel', 'Mustard', 'in', 'the',
'study', 'with', 'the', 'candlestick', '.']
```

Input: List of important words

```
['Mr.', 'Green', 'Colonel', 'Mustard', 'candlestick']
```

Input/Assumption: Cluster threshold (distance)

```
3
```

Intermediate Computation: Clusters detected

```
[ ['Mr.', 'Green', 'killed', 'Colonel', 'Mustard'], ['candlestick'] ]
```

Intermediate Computation: Cluster scores

```
[ 3.2, 1 ] # Computation: [ (4*4)/5, (1*1)/1]
```

Output: Sentence score

```
3.2 # max([3.2, 1])
```

The actual work done in `score_sentences` is just bookkeeping to detect the clusters in the sentence. A cluster is defined as a sequence of words containing two or more important words, where each important word is within a distance threshold of its nearest neighbor. While Luhn's paper suggests a value of 4 or 5 for the distance threshold, we used a value of 3 for simplicity in this example; thus, the distance between 'Green' and 'Colonel' was sufficiently bridged, and the first cluster detected consisted of the first five words in the sentence. Note that had the word "study" also appeared in the list of important words, the entire sentence (except the final punctuation) would have emerged as a cluster.

Once each sentence has been scored, all that's left is to determine which sentences to return as a summary. The sample implementation provides two approaches. The first approach uses a statistical threshold to filter out sentences by computing the mean and standard deviation for the scores obtained, while the latter simply returns the top *N* sentences. Depending on the nature of the data, your mileage will probably vary, but you should be able to tune the parameters to achieve reasonable results with either. One nice thing about using the top *N* sentences is that you have a pretty good idea about the maximum length of the summary. Using the mean and standard deviation could potentially return more sentences than you'd prefer, if a lot of sentences contain scores that are relatively close to one another.

Luhn's algorithm is simple to implement and plays to the usual strength of frequently appearing words being descriptive of the overall document. However, keep in mind that like many of the approaches based on the classic information retrieval concepts we explored in the previous chapter, Luhn's algorithm itself makes no attempt to understand the data at a deeper semantic level although it does depend on more than just a "bag of words." It directly computes summarizations as a function of frequently occurring

words, and it isn't terribly sophisticated in how it scores sentences, but as was the case with TF-IDF, this makes it all the more amazing that it can perform as well as it seems to perform on randomly selected blog data.

When weighing the pros and cons of implementing a much more complicated approach, it's worth reflecting on the effort that would be required to improve upon a reasonable summarization such as that produced by Luhn's algorithm. Sometimes, a crude heuristic is all you really need to accomplish your goal. At other times, however, you may need something more state of the art. The tricky part is computing the cost-benefit analysis of migrating from the crude heuristic to the state-of-the-art solution. Many of us tend to be overly optimistic about the relative effort involved.

Entity-Centric Analysis: A Deeper Understanding of the Data

Throughout this chapter, it's been implied that analytic approaches that exhibit a deeper understanding of the data can be dramatically more powerful than approaches that simply treat each token as an opaque symbol. But what does "a deeper understanding" of the data really mean? One interpretation is being able to detect the entities in documents and using those entities as the basis of analysis, as opposed to document-centric analysis involving keyword searches or interpreting a search input as a particular type of entity and customizing results accordingly. Although you may not have thought about it in those terms, this is precisely what emerging technologies such as WolframAlpha do at the presentation layer. For example, a search for "tim o'reilly" in WolframAlpha returns results that imply an understanding that the entity being searched for is a person; you don't just get back a list of documents containing the keywords (see Figure 5-4). Regardless of the internal technique that's used to accomplish this end, the resulting user experience is dramatically more powerful because the results conform to a format that more closely satisfies the user's expectations.

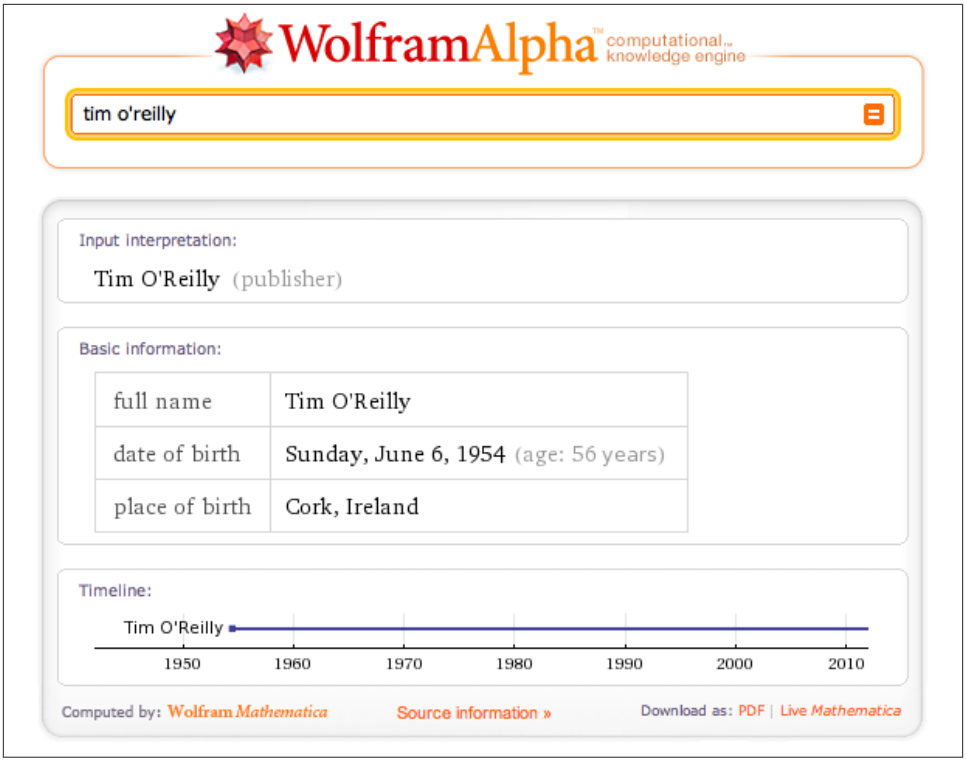


Figure 5-4. Sample results for a "tim o'reilly" query with WolframAlpha

Although it's beyond the scope of this chapter to ponder all of the various possibilities of entity-centric analysis, it's well within our scope and quite appropriate to present a means of extracting the entities from a document, which can then be used for various analytic purposes. Assuming the sample flow of an NLP pipeline as presented earlier in this chapter, one approach you could take would be to simply extract all the nouns and noun phrases from the document and index them as entities appearing in the documents —the important underlying assumption being that nouns and noun phrases (or some carefully constructed subset thereof) qualify as entities of interest. This is actually a very fair assumption to make and a good starting point for entity-centric analysis, as the following sample listing demonstrates. Note that for results annotated according to Penn Treebank conventions, any tag beginning with 'NN' is some form of a noun or noun phrase. A full listing of the Penn Treebank Tags is available online.

Example 5-8 analyzes the part-of-speech tags that are applied to tokens, and identifies nouns and noun phrases as entities. In data-mining parlance, finding the entities in a text is called *entity extraction* or *named entity recognition*.

Example 5-8. Extracting entities from a text with NLTK

```
import nltk
import json

BLOG_DATA = "O'Reilly Radar - Insight, analysis, and research about emerging technologies.json"

blog_data = json.loads(open(BLOG_DATA).read())

for post in blog_data:

    sentences = nltk.tokenize.sent_tokenize(post['content'])
    tokens = [nltk.tokenize.word_tokenize(s) for s in sentences]
    pos_tagged_tokens = [nltk.pos_tag(t) for t in tokens]

    # Flatten the list since we're not using sentence structure
    # and sentences are guaranteed to be separated by a special
    # POS tuple such as ('.', '.')

    pos_tagged_tokens = [token for sent in pos_tagged_tokens for token in sent]

    all_entity_chunks = []
    previous_pos = None
    current_entity_chunk = []
    for (token, pos) in pos_tagged_tokens:

        if pos == previous_pos and pos.startswith('NN'):
            current_entity_chunk.append(token)
        elif pos.startswith('NN'):
            if current_entity_chunk != []:

                # Note that current_entity_chunk could be a duplicate when appended,
                # so frequency analysis again becomes a consideration

                all_entity_chunks.append((' '.join(current_entity_chunk), pos))
            current_entity_chunk = [token]

        previous_pos = pos

    # Store the chunks as an index for the document
    # and account for frequency while we're at it...

    post['entities'] = {}
    for c in all_entity_chunks:
        post['entities'][c] = post['entities'].get(c, 0) + 1

    # For example, we could display just the title-cased entities

    print post['title']
    print '-' * len(post['title'])
    proper_nouns = []
    for (entity, pos) in post['entities']:
        if entity.istitle():
```

```
        print '\t%s (%s)' % (entity, post['entities'][(entity, pos)])
print
```

You may recall from the description of "extraction" in (to come) that NLTK provides an `nltk.batch_ne_chunk` function that attempts to extract named entities from POS-tagged tokens. You're welcome to use this capability directly, but you may find that your mileage varies with the out-of-the-box models that back the implementation. A discussion of improving the implementation that ships with NLTK is outside the scope of this chapter but an interesting pursuit to consider for the truly curious reader.

Sample output for the listing is presented below and conveys results that are quite meaningful and would make great suggestions for tags by an intelligent blogging platform. For a larger corpus than we're working with in this example, a tag cloud could be an obvious candidate for visualizing the data.

```
The Louvre of the Industrial Age
------------------------------- Paris (1) Henry Ford Museum (1) Vatican Museum (1) Museum (1) Tho
    Louvre (1)
```

Could we have discovered the same list of terms by more blindly analyzing the lexical characteristics (such as use of capitalization) of the sentence? Perhaps, but keep in mind that this technique can also capture nouns and noun phrases that are not indicated by title case. Case is indeed an important feature of the text that can generally be exploited to great benefit, but there are other interesting entities in the sample text that are all lowercase (for example, "chief curator", "locomotives", and "lightbulbs").

Although the list of entities certainly doesn't convey the overall meaning of the text as effectively as the summary we computed earlier, identifying these entities can be extremely valuable for analysis since they have meaning at a semantic level and are not just frequently occurring words. In fact, the frequencies of most of the terms displayed in the sample output are quite low. Nevertheless, they're important because they have a grounded meaning in the text—namely, they're people, places, things, or ideas, which are generally the substantive information in the data.

It's not much of a leap at this point to think that it would be another major step forward to take into account the verbs and compute triples of the form subject-verb-object so that you know which entities are interacting with which other entities, and the nature of those interactions. Such triples would lend themselves to visualizing object graphs of documents, which could potentially be skimmed much faster than reading the documents themselves. Better yet, imagine taking multiple object graphs derived from a set of documents and merging them to get the gist of the larger corpus. This exact technique is very much an area of active research and has tremendous applicability for virtually

any situation suffering from the information-overload problem. But as will be illustrated, it's an excruciating problem for the general case and not for the faint of heart.

Assuming a part-of-speech tagger has identified the following parts of speech from a sentence and emitted output such as `[('Mr.', 'NNP'), ('Green', 'NNP'), ('killed', 'VBD'), ('Colonel', 'NNP'), ('Mustard', 'NNP'), ...]`, an index storing subject-predicate-object tuples of the form `('Mr. Green', 'killed', 'Colonel Mustard')` would be easy to compute. However, the reality of the situation is that you're very unlikely to run across actual POS-tagged data with that level of simplicity—unless you're planning to mine children's books (something that's not actually a bad starting point to test out toy ideas.) For example, consider the tagging emitted from NLTK for the first sentence from the blog post printed earlier in this chapter as an arbitrary and realistic piece of data you might like to translate into an object graph:

> This morning I had the chance to get a tour of The Henry Ford Museum in Dearborn, MI, along with Dale Dougherty, creator of Make: and Makerfaire, and Marc Greuther, the chief curator of the museum.

The simplest possible triple that you might expect to distill from that sentence is `('I', 'get', 'tour')`, but even if you got that back, it wouldn't convey that Dale Dougherty also got the tour, or that Mark Greuther was involved. The POS-tagged data should make it pretty clear that it's not quite so straightforward to arrive at any of those interpretations, either, because the sentence has a very rich structure:

```
[(u'This', 'DT'), (u'morning', 'NN'), (u'I', 'PRP'), (u'had', 'VBD'), (u'the', 'DT'),
(u'chance', 'NN'), (u'to', 'TO'), (u'get', 'VB'), (u'a', 'DT'), (u'tour', 'NN'),
(u'of', 'IN'), (u'The', 'DT'), (u'Henry', 'NNP'), (u'Ford', 'NNP'), (u'Museum', 'NNP'),
(u'in', 'IN'), (u'Dearborn', 'NNP'), (u',', ','), (u'MI', 'NNP'), (u',', ','),
(u'along', 'IN'), (u'with', 'IN'), (u'Dale', 'NNP'), (u'Dougherty', 'NNP'), (u',',
','), (u'creator', 'NN'), (u'of', 'IN'), (u'Make', 'NNP'), (u':', ':'), (u'and', 'CC'),
(u'Makerfaire', 'NNP'), (u',', ','), (u'and', 'CC'), (u'Marc', 'NNP'), (u'Greuther',
'NNP'), (u',', ','), (u'the', 'DT'), (u'chief', 'NN'), (u'curator', 'NN'), (u'of',
'IN'), (u'the', 'DT'), (u'museum', 'NN'), (u'.', '.')]
```

It's doubtful whether a high quality open source NLP toolkit would be capable of emitting meaningful triples in this case, given the complex nature of the predicate "had a chance to get a tour", and that the other actors involved in the tour are listed in a phrase appended to the end of the sentence. If you'd like to pursue strategies for constructing these triples, you should be able to use reasonably accurate POS tagging information to take a good initial stab at it. The difficulty of this task is pointed out not to discourage you, but to provide a realistic view of the complexity of NLP in general so that you know what you're getting into and properly manage your expectations. It can be a lot of work, but the results are well worth it and can be financially lucrative as a new wave of technology emerges for understanding human language data.

All that said, the good news is that you can actually do a lot of interesting things by distilling just the entities from text and using them as the basis of analysis, as demonstrated earlier. You can very easily produce triples from text on a per-sentence basis, where the "predicate" of each triple is a notion of a generic relationship signifying that the subject and object "interacted" with one another. Example 5-9 is a refactoring of Example 5-8 that collects entities on a per-sentence basis, which could be quite useful for computing the interactions between entities using a sentence as a context window.

Example 5-9. Discovering interactions between entities

```
import nltk
import json

BLOG_DATA = "O'Reilly Radar - Insight, analysis, and research about emerging technologies.json"

def extract_interactions(txt):
    sentences = nltk.tokenize.sent_tokenize(txt)
    tokens = [nltk.tokenize.word_tokenize(s) for s in sentences]
    pos_tagged_tokens = [nltk.pos_tag(t) for t in tokens]

    entity_interactions = []
    for sentence in pos_tagged_tokens:

        all_entity_chunks = []
        previous_pos = None
        current_entity_chunk = []

        for (token, pos) in sentence:

            if pos == previous_pos and pos.startswith('NN'):
                current_entity_chunk.append(token)
            elif pos.startswith('NN'):
                if current_entity_chunk != []:
                    all_entity_chunks.append((' '.join(current_entity_chunk),
                            pos))
                current_entity_chunk = [token]

            previous_pos = pos

        if len(all_entity_chunks) > 1:
            entity_interactions.append(all_entity_chunks)
        else:
            entity_interactions.append([])

    assert len(entity_interactions) == len(sentences)

    return dict(entity_interactions=entity_interactions,
                sentences=sentences)

blog_data = json.loads(open(BLOG_DATA).read())
```

```
# Display selected interactions on a per-sentence basis

for post in blog_data:

    post.update(extract_interactions(post['content']))

    print post['title']
    print '-' * len(post['title'])
    for interactions in post['entity_interactions']:
        print '; '.join([i[0] for i in interactions])
    print
```

The results from this listing are presented below and highlight something very important about the nature of unstructured data analysis: it's messy!

```
    The Louvre of the Industrial Age
    --------------------------------
    morning; chance; tour; Henry Ford Museum; Dearborn; MI; Dale Dougherty; creator;
    Make; Makerfaire; Marc Greuther; chief curator

    tweet; Louvre

    "; Marc; artifact; museum; block; contains; Luther Burbank; shovel; Thomas Edison…

    Luther Burbank; course; inventor; treasures; nectarine; Santa Rosa

    Ford; farm boy; industrialist; Thomas Edison; friend

    museum; Ford; homage; transformation; world

    machines; steam; engines; coal; generators; houses; lathes; precision; lathes;
    makerbot; century; ribbon glass machine; incandescent; lightbulbs; world; combine;
    harvesters; railroad; locomotives; cars; airplanes; gas; stations; McDonalds;
    restaurant; epiphenomena

    Marc; eye; transformation; machines; objects; things

    advances; engineering; materials; workmanship; design; years

    years; visit; Detroit; museum; visit; Paris; Louvre; Rome; Vatican Museum; Florence;
    Uffizi Gallery; St. Petersburg; Hermitage; Berlin

    world; museums

    Museum; Makerfaire Detroit

    reach; Detroit; weekend

    day; Makerfaire; day
```

A certain amount of noise in the results is almost inevitable, but realizing results that are highly intelligible and useful—even if they do contain a manageable amount of noise

—is a very worthy aim. The amount of effort required to achieve pristine results that are nearly noise-free can be immense. In fact, in most situations, this is downright impossible because of the inherent complexity involved in natural language and the limitations of most currently available toolkits, including NLTK. If you are able to make certain assumptions about the domain of the data or have expert knowledge of the nature of the noise, you may be able to devise heuristics that are effective without risking an unacceptable amount of potential information loss. But it's a fairly difficult proposition.

Still, the interactions do provide a certain amount of "gist" that's valuable. For example, how closely would your interpretation of "morning; chance; tour; Henry Ford Museum; Dearborn; MI; Dale Dougherty; creator; Make; Makerfaire; Marc Greuther; chief curator" align with the meaning in the original sentence?

As was the case with our previous adventure in summarization, displaying markup that can be visually skimmed for inspection is also quite handy. A simple modification to Example 5-9 output as shown in Example 5-10 is all that's necessary to produce the results shown in Figure 5-5.

Example 5-10. Modification of script from Example 5-9

```
import nltk
import json

BLOG_DATA = "O'Reilly Radar - Insight, analysis, and research about emerging technologies.json"

HTML_TEMPLATE = """<html>
    <head>
        <title>%s</title>
        <meta http-equiv="Content-Type" content="text/html; charset=UTF-8"/>
    </head>
    <body>%s</body>
</html>"""

blog_data = json.loads(open(BLOG_DATA).read())

for post in blog_data:

    post.update(extract_interactions(post['content']))

    # Display output as markup with entities presented in bold text

    post['markup'] = []

    for sentence_idx in range(len(post['sentences'])):

        s = post['sentences'][sentence_idx]
        for (term, _) in post['entity_interactions'][sentence_idx]:
            s = s.replace(term, '<strong>%s</strong>' % (term, ))
```

```
    post['markup'] += [s]

filename = post['title'] + '.entity_interactions.html'
f = open(filename, 'w')
html = HTML_TEMPLATE % (post['title'] + ' Interactions', ' '.join(post['markup']),)
f.write(html.encode('utf-8'))
f.close()

print "Data written to", f.name
```

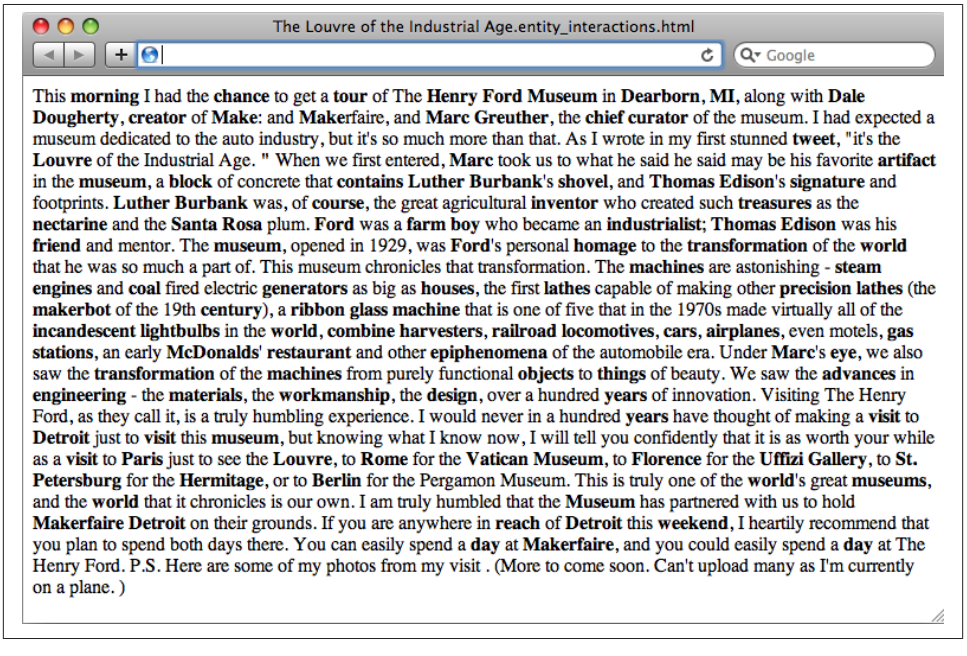


Figure 5-5. Sample HTML output that displays entities identified in the text in bold so that it's easy to visually skim the content for its key concepts

It could also be fruitful to perform additional analyses to identify the sets of interactions for a larger body of text and to find and visualize co-occurrences in the interactions. The example file *http://github.com/ptwobrussell/Mining-the-Social-Web/blob/master/python_code/introduction__retweet_visualization.py* [XXX: Update this link once the Part II code is checked in to GitHub] would make a very good starting point, but even without knowing the specific nature of the interaction, there's still a lot of value in just knowing the subject and the object. And if you're feeling ambitious and are comfortable with a certain degree of messiness and lack of precision, by all means you can attempt to fill in the missing verbs.

Quality of Analytics for Processing Human Language Data

When you've done even a modest amount of text mining, you'll eventually want to start quantifying the quality of your analytics: How accurate is your end-of-sentence detector, how accurate is your part-of-speech tagger, etc.? For example, if you began customizing the basic algorithm for extracting the entities from unstructured text, how would you know whether your algorithm was getting more or less performant with respect to the quality of the results? While you could manually inspect the results for a small corpus and tune the algorithm until you were satisfied with them, you'd still have a devil of a time determining whether your analytics would perform well on a much larger corpus or a different class of document altogether—hence, the need for a more automated process.

An obvious starting point is to randomly sample some documents and create a "golden set" of entities that you believe are absolutely crucial for a good algorithm to extract from them and use this list as the basis of evaluation. Depending on how rigorous you'd like to be, you might even be able to compute the sample error and use a statistical device called a confidence interval to predict the true error with a sufficient degree of confidence for your needs. However, what exactly is the calculation you should be computing based on the results of your extractor and golden set in order to compute accuracy? A very common calculation for measuring accuracy is called the *F1 score*, which is defined in terms of two concepts called *precision* and *recall*[4] as:

$$F = 2 \cdot \frac{\text{precision} \cdot \text{recall}}{\text{precision} + \text{recall}}$$

where:

$$\text{precision} = \frac{\text{TP}}{\text{TP} + \text{FP}}$$

and:

$$\text{recall} = \frac{\text{TP}}{\text{TP} + \text{FN}}$$

4. More precisely, F1 is said to be the *harmonic mean* of precision and recall, where the harmonic mean of any two numbers, x and y, is defined as:

$$H = 2 * \frac{x * y}{x + y}$$

   You can read more about why it's the "harmonic" mean by reviewing the definition of a harmonic number.

In the current context, precision is a measure of exactness that reflects "false positives," and recall is a measure of completeness that reflects "true positives." The following list clarifies the meaning of these terms in relation to the current discussion in case they're unfamiliar or confusing:

True positives (TP)
: Terms that were correctly identified as entities

False positives (FP)
: Terms that were identified as entities but should not have been

True negatives (TN)
: Terms that were not identified as entities and should not have been

False negatives (FN)
: Terms that were not identified as entities but should have been

Given that precision is a measure of exactness that quantifies false positives, it is defined as precision = TP / (TP + FP). Intuitively, if the number of false positives is zero, the exactness of the algorithm is perfect and the precision yields a value of 1.0. Conversely, if the number of false positives is high and begins to approach or surpass the value of true positives, precision is poor and the ratio approaches zero. As a measure of completeness, recall is defined as TP / (TP + FN) and yields a value of 1.0, indicating perfect recall, if the number of false negatives is zero. As the number of false negatives increases, recall approaches zero. Note that by definition, F1 yields a value of 1.0 when precision and recall are both perfect, and approaches zero when both precision and recall are poor. Of course, what you'll find out in the wild is that it's generally a trade-off as to whether you want to boost precision or recall, because it's very difficult to have both. If you think about it, this makes sense because of the trade-offs involved with false positives and false negatives (see Figure 5-6).

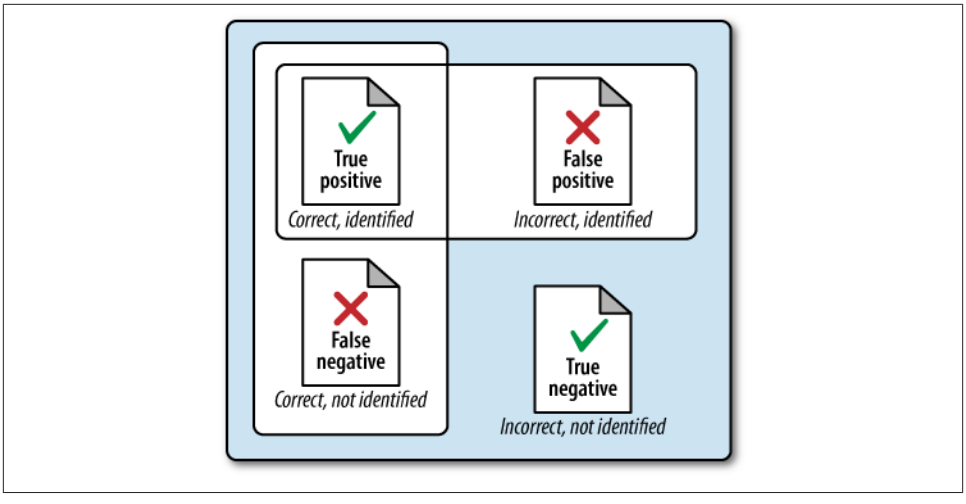


Figure 5-6. The intuition behind true positives, false positives, true negatives, and false negatives

To put all of this into perspective, let's consider the classic (by now anyway) sentence, "Mr. Green killed Colonel Mustard in the study with the candlestick" and assume that an expert has determined that the key entities in the sentence are "Mr. Green", "Colonel Mustard", "study", and "candlestick". Assuming your algorithm identified these four terms and only these four terms, you'd have 4 true positives, 0 false positives, 5 true negatives ("killed", "with", "the", "in", "the"), and 0 false negatives. That's perfect precision and perfect recall, which yields an F1 score of 1.0. Substituting various values into the precision and recall formulas is straightforward and a worthwhile exercise if this is your first time encountering these terms. For example, what would the precision, recall, and F1 score have been if your algorithm had identified "Mr. Green", "Colonel", "Mustard", and "candlestick"?

As somewhat of an aside, you might find it interesting to know that many of the most compelling technology stacks used by commercial businesses in the NLP space use advanced statistical models to process natural language according to supervised learning algorithms. Recall that a *supervised learning algorithm* is essentially an approach in which you provide training samples of the form `[(input1, output1), (input2, output2), ..., (inputN, outputN)]` to a model such that the model is able to predict the tuples with reasonable accuracy. The tricky part is ensuring that the trained model generalizes well to inputs that have not yet been encountered. If the model performs well for training data but poorly on unseen samples, it's usually said to suffer from the problem of *overfitting* the training data. A common approach for measuring the efficacy of a model is called *cross-validation*. With this approach, a portion of the training data (say, one-third) is reserved exclusively for the purpose of testing the model, and only the remainder is used for training the model.

Closing Remarks

This chapter introduced the bare essentials of advanced unstructured data analytics, and demonstrated how to use NLTK and put together the rest of an NLP pipeline and extraction entities from text. The still emerging field of understanding human language data is incredibly interdisciplinary, still quite nascent despite all of our collective attempts, and nailing the problem of NLP for most of the world's most commonly spoken languages is arguably the problem of this century (or at least the first half of it.) Push NLTK to its limits, and when you need more performance or quality, consider rolling up your sleeves and digging into some of the academic literature. It's admittedly a daunting task, but a truly worthy problem if you are interested in tackling it. There's only so much that once chapter out of any book can teach you, and hopefully one of your takeaways is that the possibilities are vast and can be very profitable if you're willing to work hard at it.

Recommended Exercises

- Adapt the code from this chapter to collect a few hundred high quality articles or blog posts from the web and summarize the content.
- Build a hosted web app with a toolkit such as Google App Engine to build an online summarization tool. (Given that Yahoo! recently acquired a company called Summly that summarizes news for readers, you may find this exercise particularly inspiring.)
- Consider using NLTK's word-stemming tools to try to compute (entity, stemmed predicate, entity) tuples, building upon the code in Example 5-9.
- Look into WordNet, a tool that you'll undoubtedly run into sooner rather than later, to discover additional meaning about predicate phrases you will encounter during NLP.
- Visualize entities extracted from text with a a tag cloud.

  Try writing your own end-of-sentence detector as a deterministic parser based upon logic you can encode as rules in if-then statements and compare it to the facilities in NLTK. Is it appropriate to try and model language with deterministic rules?
- Use Scrapy to crawl a small sample of news articles or blog posts and extract the text for processing.
- Explore NLTK's Bayesian classifier, a supervised learning technique that can be used to label training samples such as documents. Can you train a classifier to label documents as "sports", "editorial", and "other" from a small crawl with Scrapy? Measure your accuracy as an F1 score.

- Are there situations in which a "harmonic mean" between precision and recall is not desirable? When might you want higher precision at the cost of recall? When might you want higher recall at the cost of precision?